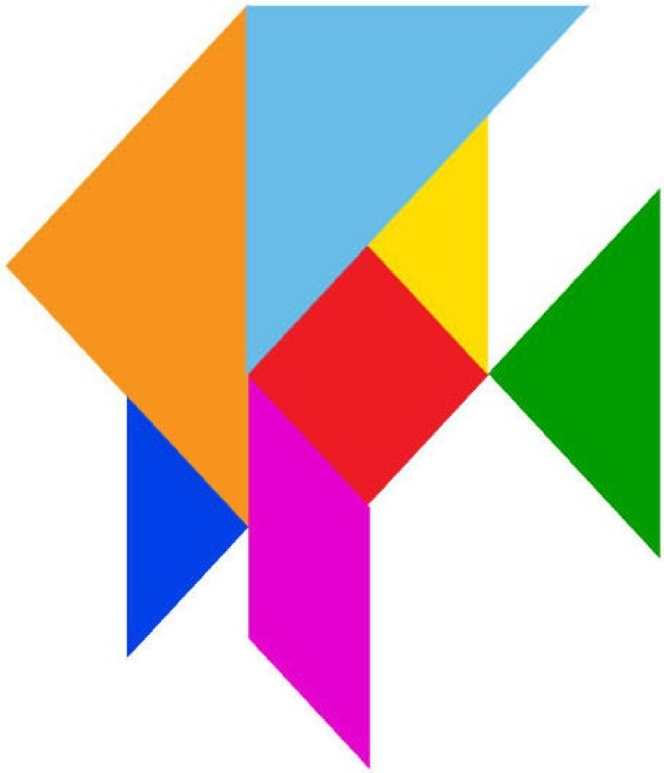
Rails Practice

Rails 实践

用 Rails 4 构建在线网店



里克的自习室 Railser.cn

目錄

写在前面

你好，我是里克，2007年开始从事 Rails 开发工作。《Rails 实践》这本书，是我第一次编写完整的教程，对我来说，它更像是对过往经验的总结。

本书通过一个在线网店程序的开发过程，带领大家了解 Rails 全貌。第一章，我们安装 Ruby 和 Rails 的开发环境，并学习如何设计项目 UI。第二章，我们讲解 Rails 中的资源含义，学习 Rails 如何实现 REST 风格架构，感受 Rails 的快捷开发。第三章，我们关注 Rails 的视图，从页面部分开始了解 MVC 框架。第四章，我们关注数据库部分，讲解 Rails 中的 M。第五章，我们在了解控制器的同时，完成我们网店的购买功能。第六章，我们学习 Rails 中的各种配置，并将它在云服务器上部署运行。

在阅读本书同时，也希望你能阅读其他 Ruby 和 Rails 的教程，博客和新闻，增加知识储备。

> 写出正确的代码是需要理由的。

阅读电子版

本书电子版为免费阅读，目前有两个指定的发布地址：

独立域名：http://rails-practice.com/

极客学院wiki：http://wiki.jikexueyuan.com/project/rails-practice/

当前版本

1.1.0

2017年2月更新说明

- 书名更新为《Rails 实践：使用 Rails 4 构建在线网店》
- 使用 Docker 作为开发调试环境，增加了 README.md，针对每一份代码有如何创建容器的说明
- Ruby 版本更新为 2.3.3，请使用 `docker pull ruby:2.3.3` 镜像创建容器
- Rails 版本统一更新为 4.2.7.1，请使用 `bundle update rails` 升级当前版本
- 调整项目代码在当前 Ruby 和 Rails 版本下可正常运行
- 更新 gems 数据源为 `https://gems.ruby-china.org`
- 去掉了 .ruby-version 和 .ruby-gemset 两个文件

- 结合大家在 issue 中反馈的问题进行调整
- 精简部分代码
- 修正 bug

本书读者

本书适合期望使用 Rails 制作 Web 网站的开发者，读者需要具备基础的 HTML，JS 和 CSS 知识，并且了解 Ruby 基本语法。你可以从未使用过 Ruby 和 Rails，这没关系，本书会带领你从安装 Ruby 环境开始，直到完成这个 Rails 项目。

在学习的过程中，我建议读者注册一个 github.com 账号，建立一个学习笔记的代码仓库（Repo）中。

本书约定

- 名词首字母大写。
- 英文缩写大写。
- 命令小写。
- 作为名词时，首字母大写，作为命令时 小写。Rails，Ruby 同。
- 专有名词不翻译。
- 专有名词按照约定书写，如 iPhone，iMac，html，js，css，php，jQuery 等等。
- 中文和英文间留有空格。
- 命令行中，当前用户操作使用 `%` 开头，root 用户操作，用 `$` 开头。

版权声明

- 以纸媒出版为目的，改写、改编以及摘抄本书的内容

读者反馈

你可以在 https://github.com/liwei78/rails-practice/issues 页面写下你的问题，也可以留下意见和建议。

示例代码

https://github.com/liwei78/rails-practice-code

你可以 `fork` 这份代码到自己的代码仓库（Repo）中，修改并提交，然后向我的代码仓库提交 `Pull Request` ，如果修改无异议，我将合并到 master 中。

作者介绍

李玮，网名里克，2007年开始从事 Rails 开发，先后经历过社会化搜索引擎 deyeb，华为生活社区百草网，电商平台等开发工作。在某网络安全公司从事产品开发。

工作之余，担任长春心语志愿者协会网络顾问，编程教练以及80学院 Rails 导师。

里克的自习室公众号

感谢

感谢北京迅思科技及 课程树 团队中的每位成员。

感谢所有关注过 里克的自习室 的朋友们。

感谢 Ruby China 社区。

第一章 Ruby on Rails 概述

课程概要：

本课程主要讲解Ruby on Rails基础知识，包括对 Rails 开发环境、Ruby版本及 Ruby 管理工具 RVM 的简单介绍， Rails 项目中的文件含义的讲解，并为即将开始的 Rails 项目设计用户界面（UI）。

知识点：

1. Rails 开发环境概述
2. Rails 中的文件概述
3. 用户界面（UI）设计

课程背景

Ruby 是一门现代，面向对象的脚本语言。 它简洁、容易理解，可以让你快速地用代码自然、清晰表达想法。让你的程序能很简单被编写并且在几个月后还能很容易读懂。Ruby on Rails 是一个 Web 应用程序框架,是一个相对较新的 Web 应用程序框架，构建在 Ruby 语言之上。它被宣传为现有企业框架的一个替代，而它的目标，简而言之，就是让生活，至少是 Web 开发方面的生活，变得更轻松。通过本课程的学习，学员能够掌握如何搭建开发环境，了解 Rails 项目中文件的含义，并通过用户界面（UI）的设计，了解项目如何交付，以及要实现的目标。

1.1 Ruby on Rails 开发环境介绍

概要：

本课时介绍了 Ruby 及 Rails 的开发环境，RVM 和 Ruby 的安装，以及操作系统平台的选择。

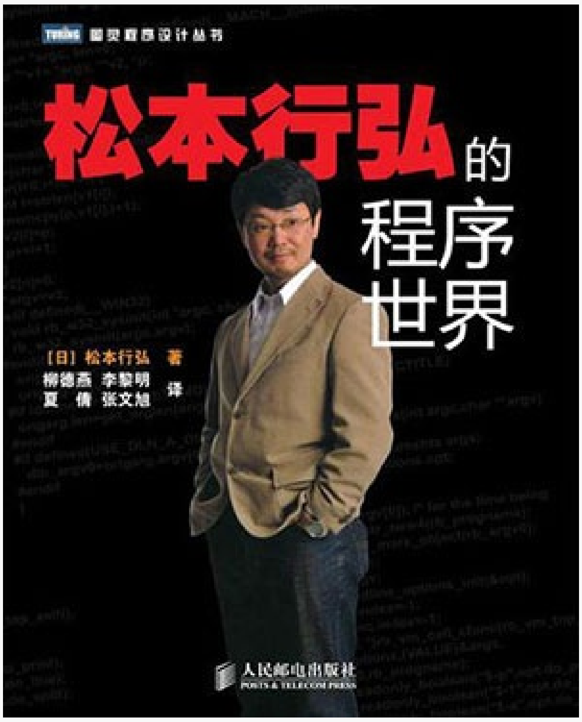
知识点：

1. RVM 安装
2. Ruby 安装
3. Rails 安装
4. 代码管理

正文

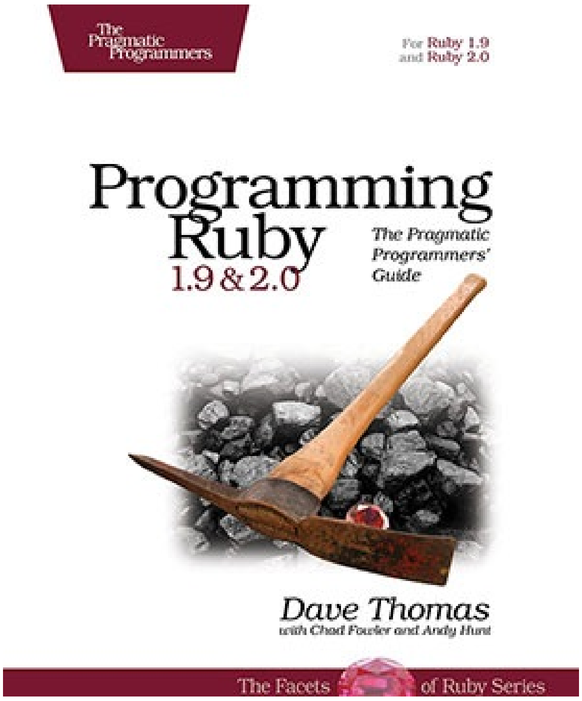
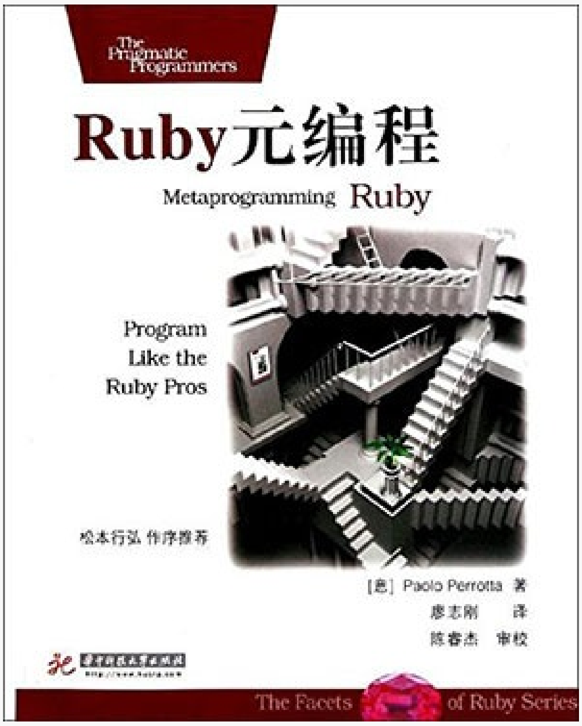
1.1.1 Ruby 简介

Ruby, 是由 松本行弘 先生在1995年正式发布的一种“面向对象编程”的脚本语言。推荐两本松本行弘的书籍。

封面	书评
	《松本行弘的程序世界》是探索程序设计思想和方法的经典之作。作者从全局的角度，利用大量的程序示例及图表，深刻阐述了Ruby编程语言的设计理念，并以独特的视角考察了与编程相关的各种技术。
	《代码的未来》是Ruby之父松本行弘的又一力作。作者对云计算、大数据时代下的各种编程语言以及相关技术进行了剖析，并对编程语言的未来发展趋势做出预测，内容涉及Go、VoltDB、node.js、CoffeeScript、Dart、MongoDB、摩尔定律、编程语言、多核、NoSQL等当今备受关注的话题。

再推荐大家几本 Ruby 开发给书，方便大家在学习 Rails 之余，更多的了解 Ruby。

封面	书名

	Ruby 编程
	Ruby 元编程



七周七语言

更多 Ruby 的介绍，大家可以查看 Ruby简介 和 20分钟体验 Ruby。

1.1.2 Rails 简介

我们使用的 Rails，就是基于 Ruby 开发的。Rails 的完整称呼是 Ruby on Rails，简称 Rails，是由 丹麦人 David Heinemeier Hansson（DHH）在2003年发布的开源 Web 框架。



图为穿着赛车服的 DHH，他和其他两队友获得了2014年勒芒24小时耐力赛GTE-Am组的冠军。

Rails 是一个基于 MVC 模式的高效的开发框架。在我刚刚接触 Rails 的2007年，很多人说不需要了解 Ruby，就可以使用 Rails 开发网站了，足见 Rails 的方便和快捷。而快速开发，也成为了 Rails 迅速获得众多开发人员喜爱的原因，众多大型网站，曾经或现在，正在使用着 Rails。Rails的受欢迎，也使得 Ruby 跻身最流行的开发语言排名前列。

注：勒芒大赛对车手是个极大的考验，FISA规定勒芒每部赛车由3名赛手分别驾驶(1980年中期以前为2名赛手)，即采用换人不换车的方法，所有的加油、换胎和维修时间都包括在24小时以内。最后，行驶里程最多的赛车获胜，一般一昼夜下来，成绩最好的赛车行驶的里程将近5000公里。每人连续驾驶时间不超过4小时，主车手总驾驶时间不超过14小时。勒芒环行跑道全长13公里，其中绝大部分是封闭式的公用高速公路，赛车在其2/3的路段上时速达370km/h左右，C组车一般只用3分钟左右的时间就能跑完一圈的路程。在跑道上有一段约6km的直路，赛车在这段路上飞速驶过，速度达到390km/h。

1.1.3 Ruby 安装

在安装 Rails 前，我们先来安装 Ruby 环境。这里，我们使用 rvm 这个工具。

注：以下安装及后续开发是在 Mac 系统上进行的，Windows 系统可以选择 rubyinstaller。但是在 windows 开发 Rails 程序会遇到众多问题，建议大家安装虚拟机或者 Linux 双系统进行开发。

RVM 是 Ruby 管理工具，可以方便的安装、管理、切换多个 Ruby，管理 Gemset。

安装 RVM 的命令是：

```
curl -sSL https://get.rvm.io | bash -s stable
```

如果你已经安装了 RVM，可以用这个命令升级到最新的 stable 版本：

```
rvm get stable
```

在有的操作系统中，会给出这个提示：

```
  * To start using RVM you need to run `source /home/webmaster/.rvm/scripts/rvm`
    in all your open shell windows, in rare cases you need to reopen all shell windows
.
```

这是你可以运行提示中的命令，`source /home/webmaster/.rvm/scripts/rvm`，或者退出当前登录 shell，再次登入。

我们在当前开发用户中安装 RVM，不必切换到 root 用户下。在生产服务器（Poduction）中，可以使用专门的项目管理用户，并具备 sudo 权限。我们在后面部署章节里会详细介绍。

安装完 RVM 后，我们可以使用 `rvm -v` 查看版本。我们使用 `rvm list known` 这个命令，可以查看可安装的 Ruby 版本：

```
% rvm list known
[ruby-]1.8.6[-p420]
[ruby-]1.8.7[-head] # security released on head
[ruby-]1.9.1[-p431]
[ruby-]1.9.2[-p330]
[ruby-]1.9.3[-p551]
[ruby-]2.0.0[-p598]
[ruby-]2.1.4
[ruby-]2.1[.5]
[ruby-]2.2.0
[ruby-]2.2-head
ruby-head
```

我们的课程里，将使用2.2.0这个版本：

```
rvm install 2.2.0
```

我们可以查看当前安装的 Ruby 版本：

```
% rvm list
=* ruby-2.2.0 [ x86_64 ]
```

如果你已经安装了其他版本的 Ruby，可以通过 `--default` 参数，设置 RVM 默认使用的 Ruby 版本：

```
rvm use 2.2.0 --default
```

看一下我们的 Ruby 版本：

```
% ruby -v
ruby 2.2.0p0 (2014-12-25 revision 49005) [x86_64-darwin13]
```

1.1.4 Rails 安装

安装 Rails 前，我们先创建一个 Gemset。Gemset 是一个独立的 Gem 集合，可以为每个项目设置自己的 Gemset，而不会相互干扰。：

```
rvm gemset create rails4.2
rvm use 2.2.0@rails4.2 --default
gem install rails -v 4.2.0 --no-document
```

注： `--no-document` 会跳过安装 ri 和 rdoc 文档，可以减少安装时间。

注：在一些系统环境中，还需要先安装 bundler，它的命令是 `gem install bundler` 。Bundler 是Ruby 跟踪和安装 Gem 的工具，它的官网在这里 http://bundler.io/ 。

在后面代码开发中，我们将继续使用 Ruby 2.2 和 Rails 4.2 版本。

这里有一份 RVM实用指南 供大家参考。

1.1.5 操作系统

Ruby 和 Rails 的开发环境，可以在多个操作系统上安装，你可以选择 Mac 作为开发平台，也可以使用 ubuntu 等 linux 系统，作为开发和生产环境部署平台。windows 系统可以作为开发平台使用。

1.1.5 代码管理

在正式进入我们的教学前，请先熟悉一下 git 的简单操作（中文版），Git是一个开源的分布式版本控制系统，用以有效、高速的处理从很小到非常大的项目版本管理。

我们的代码是放到 github 上的，你可以 clone 下来我们的代码，在本地调试。另外，你也需要准备好自己的编辑器。

github是一个打开托管平台，也是一个开发者的互动社区，你可以在上面阅读大量的开源代码，比如 Ruby，Rails，还有我们每一个章节的 代码。

墙裂建议你注册一个 github 的账号，把你学习的代码和经验总结放到上面去。代码可以创建代码仓库（repo），学习经验可以创建 github 的 wiki 页面，或者使用 markdown 来编写。对于一些实用的代码片段，可以使用 gist 保存。

阅读

我推荐大家阅读 Rails 入门 介绍，它的 中文 内容在这里。

1.2 Rails 文件简介

概要：

本课时介绍 Rails 项目创建后的文件含义，介绍Rails 项目中的三种运行环境， Gemfile 及 Gem，以及 Rake 任务等。

知识点：

1. 文件含义
2. 运行环境说明及配置
3. Gem 和 Gemfile
4. Rake 任务

正文

1.2.1 基础文件介绍

本节开始前，我们先使用一个命令，创建 Rails 项目。或许你已经知道了，它就是：

```
rails new shop
```

提示：如果你已经安装了其他版本的 Rails，那么该命令使用最新的版本创建项目，如果指定其他版本，可以这样来写

```
rails _4.1.5_ new shop
```

如果你想查看已经安装的 Rails 有哪些版本，可以使用 `gem list | grep rails` 。

好了，我们来看一下 new 为我们创建了哪些文件。

app 文件夹

我们的业务逻辑文件存放地，在后面的教程里，我们会经常为它增加内容，到时会详加介绍。

config 文件夹

这里存放的是 Rails 的配置文件。首先，打开 environments 文件夹，我们可以看到三个文件，这分别对应 Rails 的三种运行环境，我们开始时候使用的是 development 环境，运行测试时是 test 环境，当我们把代码部署到服务器上，正式上线的时候，使用的是 production 环境。

Rails 允许我们分别为三种环境做不同的设置，比如，production 中 `config.assets.digest = true` ，而开发环境可以设为 `config.assets.digest = false` 。

Rails 还为我们提供了 I18n 的管理功能，这里还只有 en.yml 一种语言包，后面的课程里，我们将详细介绍 I18n 功能，并添加简体中文和英文语言包。

database.yml 中配置了数据库信息。Rails 默认使用 sqlite 数据库作为开发使用。我们也可以更改它，在 new 的时候这样做： `rails new shop -d mysql|oracle|postgresql|...`

routes.rb 是我们的路由文件，一个非常重要的文件，我们下一个章节将从它开始介绍 Rails 的诸多优秀设计。

secrets.yml 中的配置分别对应三种运行环境，它是用来加密我们的 session 的。

db 文件夹

如果你使用的是 sqlite 数据库，那么你会发现它存放在这里。sqlite 是一种小型的便于开发环境使用的数据库。在生产环境（production）的数据库，比如 mysql，postgres 等数据库文件，是不需要放到这里的。

migrate 文件夹中，存放的是我们的数据库迁移文件，下一章我们会经常看到它。

这里还有一个 seeds.rb 文件，可以用它来为项目创建一些初始数据。

lib 文件夹

lib，在我们开发 Rails 项目是，会经常的扩展一些功能，而这些功能具有复用的特点时，可以把代码放到 lib 中。

这里我想到了 Rails 的一条哲理：Convention Over Configuration，约定优于配置。我们扩展的功能文件，可以放到任何可被夹在的目录下，但是，那违背了 Rails 的这条哲理。

log 文件夹

这里存放的是日志文件，我们可以看到它对应了上面的三种运行环境，Rails 把每一种运行环境的 log，都单独的存放。

public 文件夹

这里存放的是静态文件，比如图片，html，还有编译好的 js，css 等。

test

这里是测试文件，我们编写项目的同时，也会带领大家编写对应的测试代码。所以我们后面会经常的用到它。

不过，我们使用 Rspec 进行测试，测试文件放到 spec 文件夹里。

vendor 文件夹

这是第三方代码库，比如我们 clone 下来的 gem，下载的 js 库等。

Gemfile 文件

在之前的讲解中，我们经常提到 Gem。

Gem，是 Ruby 编写的代码库的发布包。一个 Gem 中还可以包含了其他一些 Gem，比如，Rails 就是个 Gem，其中还包含了 activerecord，activesupport 这些 Gem。可以说，Rails 就是一大堆 Gem 的集合。

Rails 是通过 Gemfile 文件，来管理众多 Gem 的。

打开 Gemfile，可以看到我们的项目使用了 Rails 4.2.0 这个版本的 Gem，使用了 sqlite3 这个数据库，以及其他的一些 Gem，这都是 Rails 4.2.0 默认使用的。

我们是可以修改这个文件，每次修改之后，我们需要 `bundle install`，它会把 Gem 的版本号和互相间的依赖关系重新的配置一遍，并且会自动的更新 Gemfile.lock 这个文件，然后安装 Gefile.lock 中声明的 Gem。

所以，即便我们使用不同的开发机器，只要 Gemfile.lock 相同，我们就会安装相同的 Gem，以保持每个开发机器使用相同的开发和运行环境。

Rakefile

Rails 为我们提供了很多便捷的 rake 任务，我们通过 `rake -T` 可以看到，如果加上 `rake -T -D`，可以看到详细的说明。当然，我们页可以自己编写 rake，把它们放到 lib/tasks/里面，扩展名是 .rake。

README.md 文件

为你的项目写一份 Readme 是很有帮助的，你有注意到.md 这个格式么？它是 markdown 格式，目前最流行的书写格式，本书也是用 markdown 写成的。

英文 介绍在这里，不过我更愿意看这里。

我创建的代码，可以在这里找到：https://github.com/liwei78/rails-practice-code/tree/master/chapter_1/shop

1.2.2 安装 Gem

安装 Gem 时，Ruby（注意，是 Ruby）使用的是 bundler 这个工具。它的官网在这里：http://bundler.io/

在我们配置 Gemfile 时，经常遇到一些配置语法，这里把常见的介绍下：

```
source 'https://rubygems.org'
# source 'http://ruby.taobao.org' # 我们也可以使用 taobao 这个安装源，不过一些 Gem 不存在时，还是要使用 rubygems 官方源的。
```

```
gem 'xxx', '~>2.0.3' # ~> 这个写法表示当前版本大于等于2.0.3，小于2.1版本
gem 'xxx', '~>2.1'   # ~> 这个写法表示当前版本大于等于2.1，小于3.0版本
```

```
gem 'my_gem', '1.0', :source => 'https://gems.example.com' # 我们可以指定自己的 source 源
```

```
gem 'nokogiri', :git => 'https://github.com/tenderlove/nokogiri.git', :branch => '1.4'
 # 也可以指定 Github 地址和分支
```

```
gem 'extracted_library', :path => './vendor/extracted_library' # 我们可以从 vendor 文件夹中安装一个 Gem。
```

```
# 我们可以为运行环境指定一个 group，比如，在 development 和 production 环境中，将不加载 rspec 这个 Gem，它只需要在 test 环境下工作。
group :test do
  gem 'rspec'
end
```

翻看 http://bundler.io/gemfile.html，查看更多 Gemfile 的介绍。

1.2.3 运行 Rake 任务

Rake 是一个 Ruby 实现的类似 make 的工具程序。任务（Tasks）是由 Ruby 代码编写的。这么讲有些抽象，我们看看 Rails 为我们提供的几个 Rake 任务：

```
rake db:create   # 创建数据库
rake db:migrate  # 更新数据库，更新的文件来自 db/migrate/
rake db:seed     # 执行 seed.rb 文件的内容，通常是创建一个默认的数据。
rake db:drop     # 删除数据库
```

上面这些命令，是在 development 环境下执行的，如果要在 production 下执行呢？

```
RAILS_ENV=production rake db:migrate
```

另一个常用的，是

```
rake routes
```

它会列出我们所有定义的路由（routes）列表。

你也可以自己编写一个 Rake 任务，放到 lib/tasks/中，扩展名为 .rake。

1.3 用户界面（UI）设计

概要：

本课时介绍 Bootstrap，以及 Bootswatch UI，通过 Gem，将 UI 文件安装到 Rails 项目中。介绍项目 UI 设计思路及工具。

知识点：

1. Bootstrap 介绍
2. Bootswatch 工具及 Gem
3. mybalsamiq 工具

更新说明

在新书编写之际，对当前内容作了一次升级。原代码编写于2015年，在此之后有不少版本的变动和 bug 的修复，导致当时的一些代码无法正确运行。

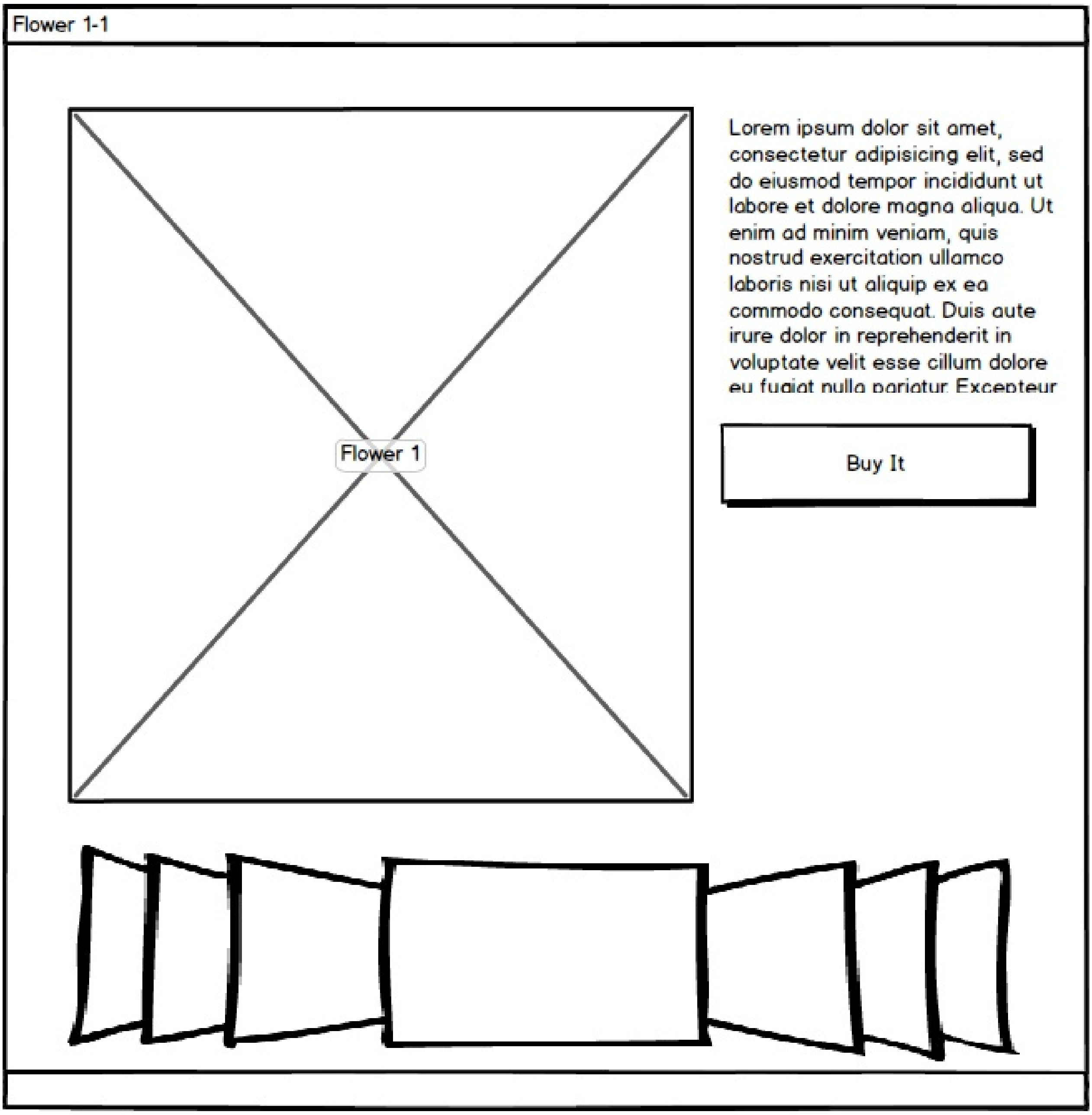
本节中，修复了一些大家反馈的问题，如果你在调试时遇到问题，可以使用 `bundle update rails` 升级到 4.2.7.1，并且 `bundle update sass-rails` 升级到4.0.5 版本，经过我的调试，这是没有异常的。

正文

1.3.1 Bootstrap

大家好，在编写我们项目代码之前，我先讲一个大约十年前的事情。2005年创业初期，为客户制作网站，有一次，一个客户找到我们，说要开发一个卖花的网站，因为新品即将上市，所以有一些急。于是，我们给出了厚厚的几页所谓的“设计方案”。但是客户几分钟就否定了，说：“我们的项目很简单，只需要购买者看到新品就可以，可以预定，我们货到付款”。于是，我们把多余的设计去掉后，之前那份设计方案只剩下三分之一了。但是客户又很快否定了我们的方案，说：“我能先看看样子么？”

于是，我们让设计师设计好了几个样子，交给客户，客户又把我们否定了，而且显得不耐烦。他抓取一张纸和一支铅笔，在纸上画出了他要的样子。什么样子呢？



在稍后商讨细节后，我们很快完成了代码功能。

这件事情给我的启发是：

> 代码之前，先看到样子

在客户画出样稿前，我们并不知道新品只有几种，而且这个网站只放置新品。它所突出的是在线预定和货到付款，即宣传了新品，又使用了另一种贴近新品的设计风格。

回到我们的例子，我们还没开始 Rails 项目之前，要先为它设计一个样子出来。有些难度么？我们先讲一个接下来要帮助我们的前端设计框架：Bootstrap。

> Bootstrap，来自 Twitter，是目前最受欢迎的前端框架。Bootstrap 是基于 HTML、CSS、JAVASCRIPT 的，它简洁灵活，使得 Web 开发更加快捷。[1] 它由Twitter的设计师Mark Otto和Jacob Thornton合作开发，是一个CSS/HTML框架。Bootstrap提供了优雅的HTML和CSS规范，它即是由动态CSS语言Less写成。Bootstrap一经推出后颇受欢迎，一直是GitHub上的热门开源项目，包括NASA的MSNBC（微软全国广播公司）的Breaking News都使用了该项目。百度百科

先给大家 Bootstrap 的 官网，这里可以找到它的 源代码，这里有中文的学习资料 Bootstrap 中文网。

在读 Bootstrap 起步 之前，我先介绍下它的特点：

- 一致的设计风格，丰富的Web组件，下拉菜单、按钮组、按钮下拉菜单、导航、导航条、面包屑、分页、排版、缩略图、警告对话框、进度条、媒体对象等
- 支持多个主流浏览器
- HTML5和CSS3开发
- 在jQuery的基础上设计，兼容大部分jQuery插件
- 平台自适应，即便在手机，pad 打开网站也没问题

> 什么？ie6？请阅读10年前的教程吧，如果还能找到的话。

在 这里，你可以很快看到 Bootstrap 的模样了。接下来的章节里，我们将按照这个样子，设计我们的 shop。

```
rails new shop
```

好的，我们给它添加个几个 gem。

```
gem "therubyracer"
gem "less-rails"
gem "twitter-bootstrap-rails"
```

然后，运行

`bundle install`

之后，我们给出一个新的命令，scaffold：

`rails g scaffold product name price:decimal description:text`

scaffold 命令我们将在下一章详细介绍，这里，我们创建了一个资源，Product。

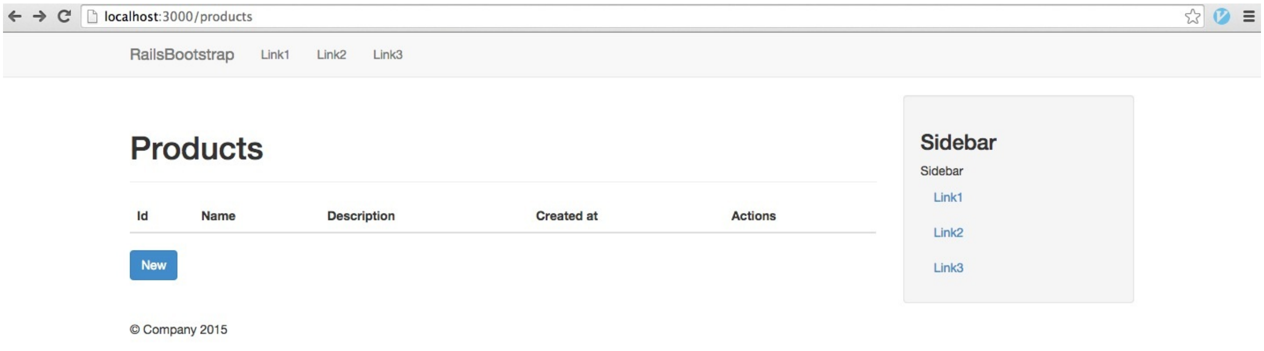
然后，我们继续运行以下几个命令

```
# 更新 db 解构
rake db:migrate
# 安装 bootstrap 文件
rails generate bootstrap:install
# 创建一个 layout
rails g bootstrap:layout
# 创建资源模板
rails g bootstrap:themed Products
```

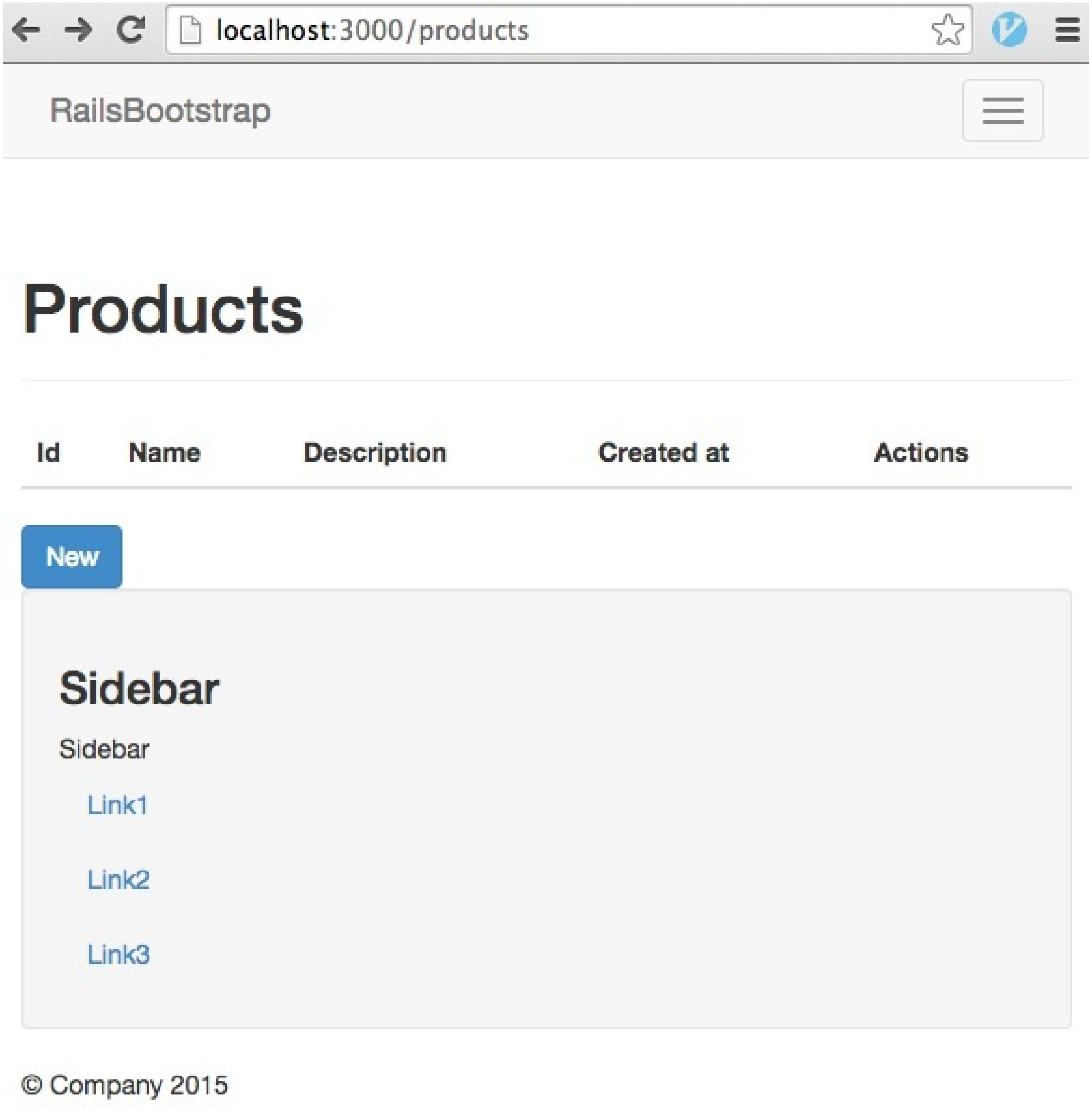
是不是还有不熟悉的命令，我们后面的章节详细介绍他们，现在，你可以运行

`rails s`

来启动 Rails 项目了，访问 `http://localhost:3000/products` ，你会看到这个页面，它就是 Bootstrap 风格的页面了。



把它缩小看看



是的，即便你用手机来访问它，也不会让页面乱掉。

我们用的是这个 gem，你可以详细的看看它的文档。

https://github.com/seyhunak/twitter-bootstrap-rails

1.3.2 Bootswatch

> 是不是太千篇一律了呀？

的确，大多数项目开始的时候都是一个样子，是件让人气馁的事情。我们来给它增加点不同。

这里再介绍一个可以帮助我们的项目，Bootswatch

我们在刚才的 Gemfile 中，再添加两个 gem：

```
gem 'twitter-bootswatch-rails'
gem 'twitter-bootswatch-rails-helpers'
```

在我们的项目中，运行下面的两个新命令：

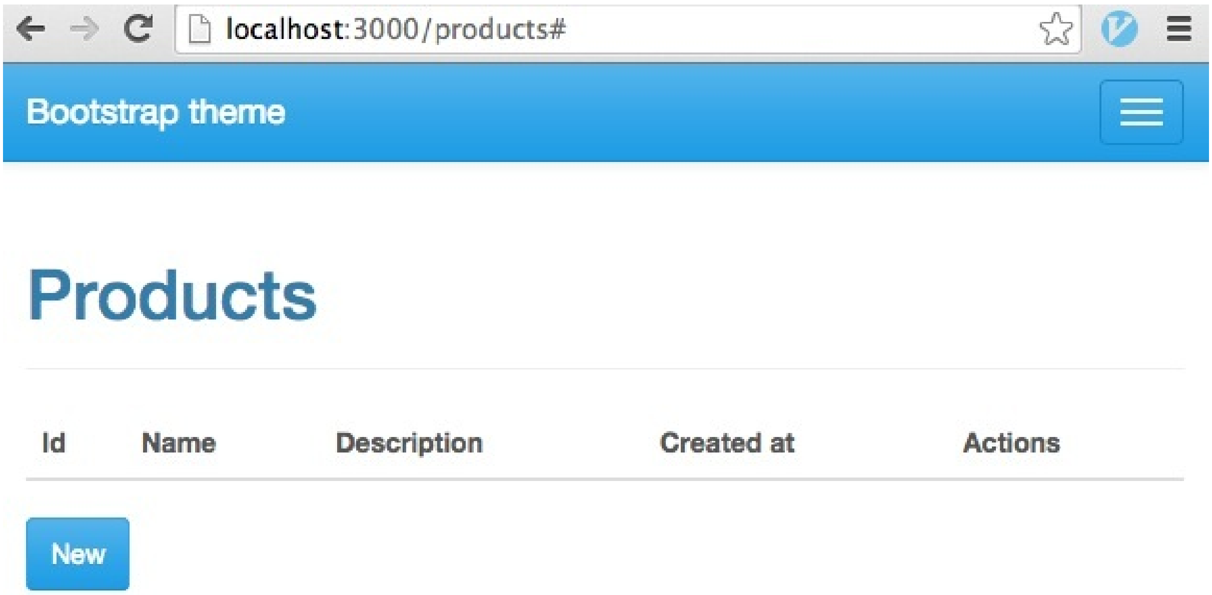
```
rails g bootswatch:install cerulean # 安装该 theme 的基础文件
rails g bootswatch:import cerulean # 导入一个线上的 theme 的变量文件
```

注：我们使用的 Gem 中，会存在 bug，或者，版本更新导致的 Gem 不匹配， 也会引起 Bug。 这时候，我们可以帮助作者改进它。当然，你要先十分确定，它是一个 Bug！

我们修改一下 `application.css` 中的引用：

```
 *= require cerulean/loader
 *= require cerulean/bootswatch
```

我们可以看到



当然，事情并未像上面写的如此容易。我在为大家写这段代码的时候，就遇到了很多问题，还好，都一一解决了。你可以到 这里 看到我调试好的代码。

在 这里，我为大家选择了三套不同的 bootswatch theme，大家可以练习。

Bootswatch-rails 的代码在这里：

https://github.com/scottvrosenthal/twitter-bootswatch-rails

> Rails 和 Ruby 一样，是为有经验的开发者准备的。

作为初学者，Rails 的确会为大家提出很多问题，有些问题会占用大量的时间，让人失去耐心。虽然开发了很多年的 Rails 项目代码，我也会经常遇到各种问题。所以，请大家耐心，让我们一起 弄清思路，慢慢解决。

1.3.3 UI 设计

本节，让我们轻松一下。

你有注意到 1.3.1 里的那张图么？对了，它是用 www.mybalsamiq.com 画的。

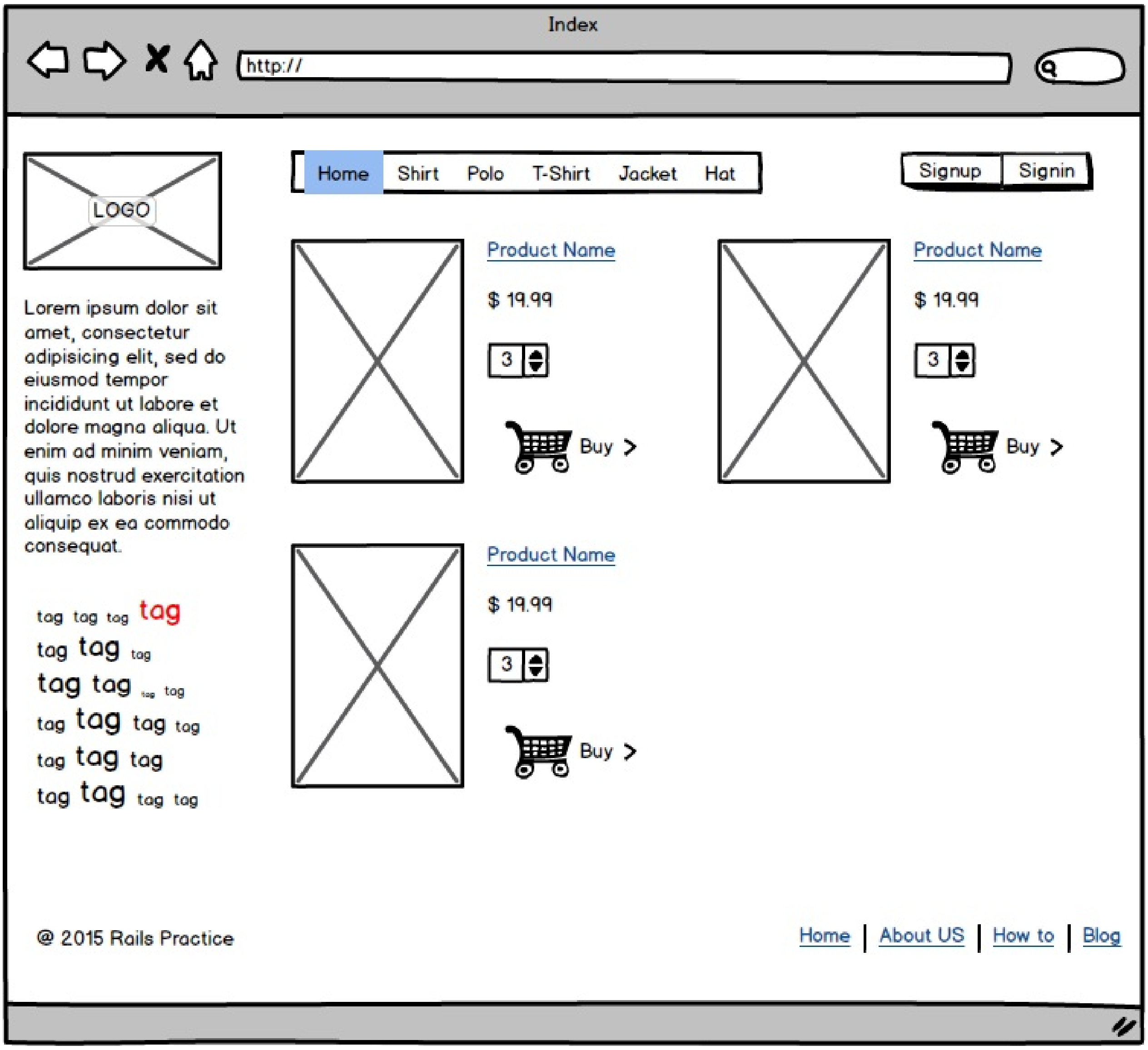
让我们继续为即将开始的 shop 项目，画几张图吧。

首先，我们想想，我们需要哪些页面。

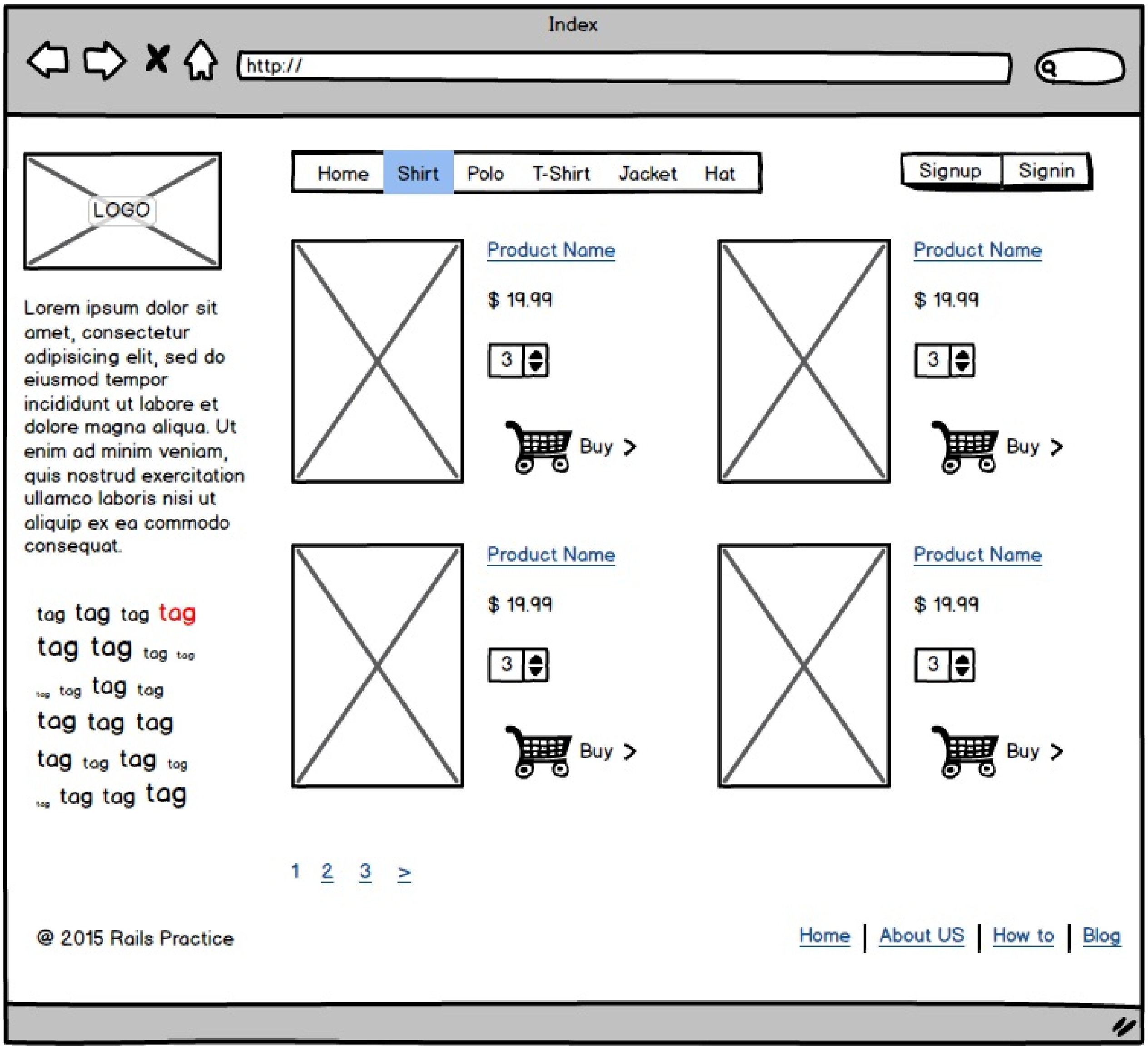
1. 首页，列出我们推荐的商品（Product）
2. 列表页，根据选择的分类，列出该分类下的商品
3. 展示页，查看每一个商品

好的，我们画出心里构思好的页面。

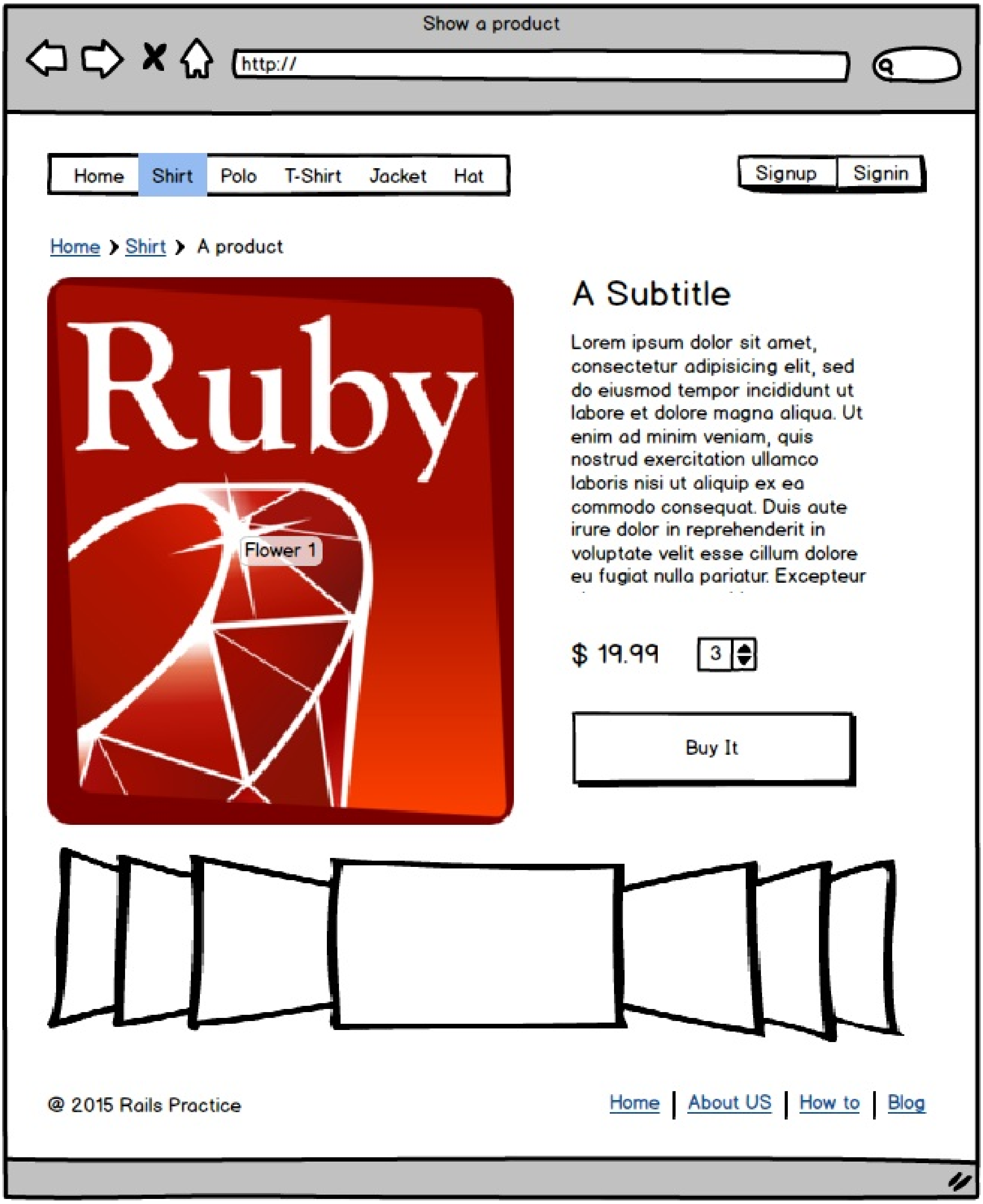
我们的首页


Index
http://
LOGO
Home
Shirt
Polo
T-Shirt
Jacket
Hat
Signup
Signin
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
tag tag tag tag
tag tag tag
tag tag tag
tag tag tag tag
tag tag tag
tag tag tag tag
Product Name
$ 19.99
3
Buy >
Product Name
$ 19.99
3
Buy >
Product Name
$ 19.99
3
Buy >
@ 2015 Rails Practice
Home
About US
How to
Blog

我们的列表页


Index
http://
LOGO
Home
Shirt
Polo
T-Shirt
Jacket
Hat
Signup
Signin
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
tag tag tag tag
tag tag tag tag
tag tag tag tag
tag tag tag
tag tag tag tag
tag tag tag
Product Name
$ 19.99
3
Buy >
Product Name
$ 19.99
3
Buy >
Product Name
$ 19.99
3
Buy >
Product Name
$ 19.99
3
Buy >
1 2 3 >
@ 2015 Rails Practice
Home | About US | How to | Blog

我们的展示页



我想讲几个我们设计上的细节。

- 首页，我们展示的是属性为置顶（top = true）的商品。
- 列表页，我们有商品分页。
- 展示页，当前分类和导航中的分类是选中状态。

当然，我们的原型设计不止这三张图，在后面的代码阶段，我们将会根据需要，再设计其他的页面。

下一节，我们将使用 `scaffold` 这个命令，来创建我们的第一个资源。我们下一节再见。

第二章 Rails 中的资源

课程概要：

本课程讲解 Rails 中的资源，通过 scaffold 命令创建资源，并对资源文件的类型、REST 风格设计及路由文件进行讲解，理解 Rails 如何在 REST 架构下如何进行资源的管理。

知识点：

1. scaffold 命令
2. REST
3. routes

课程背景

Rails 是 REST 风格的 web 开发框架，通过本课程的学习，可以理解 Rails 是如何通过对资源的管理，实现 REST 架构的。

2.1 应用 scaffold 命令创建资源

概要：

本课时详细介绍 Rails 中的命令，以及使用 scaffold 命令创建的资源文件，如 erb 文件，测试文件，sass 文件等。

知识点：

1. scaffold
2. sass/scss
3. coffeescript
4. erb
5. rspec

正文

2.1.1 rails 命令

上一节，我们用 `new` 创建了一个 Rails 项目，并且使用了下面的命令，创建了商品（Product）这个资源：

```
% rails g scaffold product name price:decimal description:text
```

g 是 generate 的缩写，Rails 还为我们提供了几个类似的命令：

```
% rails -h
generate    生成资源文件 (简写 "g")
console     运行调试控制台 (简写 "c")
server      运行 Rails 服务 (简写 "s")
dbconsole   运行数据库调试控制台 (简写 "db")
new         创建新的 Rails 项目。 你也可以 "rails new ." 它会在当前目录下创建
destroy     删除 "generate" 创建的文件 (简写 "d")
plugin new  创建一个 plugin
runner      在 Rails 环境下，执行一段 Ruby 代码
```

我们用 generate 可以创建资源，同样也可以用 destroy 删除这个资源，比如：

```
% rails destroy scaffold product
```

我们已经使用 server 命令运行了项目，但是，有时候我们不一定要在 web 页面上做操作，为了方便调试，我们可以已进入到互动终端，也就是 console 中：

```
% rails console
> Product.first
Product Load (0.2ms)  SELECT  "products".* FROM "products"  ORDER BY "products"."id" A
SC LIMIT 1
 => nil
> exit
```

在 console 里，我们可以方便的操作数据库，下一章我们会重点讲解数据库部分。

和 console 类似，dbconsole 可以进入到数据库的互动终端里，具体的命令取决去使用的哪个数据库，rails 只是为我们提供了一个方便的连接数据库方式。

plugin 命令可以为一个独立的功能创建专属的代码，并且存在与该 Rails 项目中，在 Rails 3 之后，越来越多的功能使用 gem 来实现，plugin 较少使用了。

和 console 的交互式操作不同，runner 可以执行一段代码，相同的是，它们都拥有当前项目完整的 Rails 环境。我们可以使用执行一个文件，比如：

```
% rails runner lib/somefile.rb
```

在这个 file 里，可以实现一些功能，这和 rake 的实现方式较接近。

2.1.2 scaffold 命令

回到我们经常使用的 generate 命令，先查看帮助文档：

```
% rails generate -h
```

我们看到，generate 可以创建很多类型的文件，比如 model，controller，assets 文件等，这些都是一个 Rails 资源所需要。我们可以分别执行 generate 命令，也可以把它们一次都执行，这就是 scaffold。

scaffold 的中文称呼是 “脚手架”，个人觉得它不是很形象，如果称它为“一大堆 generator（生成器） 的集合”，似乎形象很多。Rails 为我们提供了一些 generator，我们也可以编写自己的 generator。

再来看看 scaffold 的语法结构：

```
% rails g scaffold [资源名] [属性列表] [选项]
```

为了使我们的网店更接近实际应用，我们再增加一个资源：商品类型(Variant)

```
% rails g scaffold variants product_id:integer price:decimal{'8,2'} size
```

variants 是资源的名称，它可以是单数，我们创建商品时用的就是单数形式。属性列表里，属性名称和属性的类型，使用 : 来分开，默认的类型是 string，所以 color 的后面没有声明它是什么类型，那么它就是 string 类型。

当我们创建价格类型的时候，decimal 可以增加两个具体参数：precision 和 scale，每个数据库的默认值是不同的，我们可以查看这里：

RMDB 的decimal 默认值

我们打开 `config/routes.rb` ，可以看到这样两行代码已经添加了：

生成文件的时候，我们注意到这一行：

```
...
resources :products
resources :variants
...
```

这就是我们定义的资源。其实，我们说 URL 中的 R 就是这个资源 Resource 的意思。我们可以这样理解 Rails：

> Rails 是从管理资源开始的。

我们还可以配置 scaffold，让它跳过一些不必要的文件，配置写在 `config/application.rb` 中：

```
class Application < Rails::Application
  ...
  config.generators do |cfg|
    cfg.stylesheets     false
    cfg.javascripts     false
    cfg.helpers         false
  end
```

这样，scaffold 命令就不再创建 helper，css，js 文件了。但在我们学习初期阶段，先不这么做。

2.1.3 sass/scss

创建的文件中，我们看到了 `.scss` 的文件，其实，它是 `sass` 文件，一种 css 的预处理文件，它的后缀有两种：`.scss` 和 `.sass`。`scss` 语法更接近 css 本身，你可以直接粘贴 css 来使用。sass 语法更加简洁，它去掉了 `;` 和 `{}` 这些符号，并且使用空格，作为语法缩进。

使用 sass，可以使用预定义变量，使用语法嵌套，代码混入等多种编程风格的代码，编写 css，并且在编译成 css 文件的过程中，还可以进行语法检查。

sass 和 scss 写法上的不同，可以在 http://sass-lang.com/guide (中文文档)看到。

如果你想在两种文件间转换，可以使用这个命令：

```
# Convert Sass to SCSS
% sass-convert style.sass style.scss

# Convert SCSS to Sass
% sass-convert style.scss style.sass
```

SASS用法指南 一文里有更详细的介绍。

Rails 默认使用的是 sass，这里是它 github 的地址，

在我们的 css 文件中，经常会使用图片文件，比如 background-image 属性，但是我们的图片是放在 assets 文件夹中的，我们可以有三种方式来使用图片。

第一种，直接粘贴图片地址，比如：

```
background-image: url("/assets/logo.png");
```

这是很不好的，它不能使用 digest 方式来使用图片资源，也不够灵活。

第二种，将图片文件放到 public 下，比如：

```
background-image: url("/images/logo.png");
```

这需要我们在 public 下建立一个 images 文件夹来管理图片文件，也不能使用 digest。

第三种，直接使用 sass-rails 提供的辅助方法：

```
background-image: asset-url('logo.png');
```

当然，我也见到过第四种方法，使用 erb 来重构 sass，文件可能是这样的，`style.css.scss.erb`，这样就可以在 scss 里插入 erb 的语法：

```
background-image: url(<%= asset_path 'logo.png' %>);
```

在 Rails 里是可以这么写的，它会先解析 erb 文件，再解析 sass 文件，生成 css。但是我不建议这么处理问题，在我们使用一个不熟悉的方法解决问题时，应该多耐心看一看它的文档。不知道这里给出的众多链接，大家是否查看了，他们都是对内容很好的补充。

和 sass 一样，Less 也是 css 的预编译工具，如果你留意 bootstrap 的介绍，它的 css 文件就是用 less 编写的。

2.1.4 coffeescript

`.coffee` 是 js 的预处理文件，它是用 coffeescript 编写的。学习它很简单，只要看看http://coffeescript.org/ 就可以了，中文在 http://coffee-script.org/。

scss 和 coffeescript 的目标，是让代码更简洁，易维护。预处理还可以帮你检查语法上的错误。

在我们安装完 bootstrap 后，会给出一个 coffee 文件：

```
jQuery ->
  $("a[rel~=popover], .has-popover").popover()
  $("a[rel~=tooltip], .has-tooltip").tooltip()
```

2.1.5 erb

最后，我们说说 erb。

erb 是 Ruby 的标准库（Standard Library）之一， 它允许是把 Ruby 代码签入到 html 中。

一个简单的例子，我们进入到 irb 中：

```
% require "erb"
% name = "Ruby"
% ERB.new("My name is #{name}").result
=> "My name is Ruby"
```

好了，文件都介绍完了，我们看一下效果吧，我们使用下面的命令：

```
% rake db:migrate [1]
% rails s [2]
```

- [1] 更新数据库
- [2] 启动 Rails 服务，s 是 server 的简写

访问 `http://localhost:3000/products` ，试试上面的按钮，体验一下如何增加，修改，删除一个商品（Product）吧。

2.1.6 测试

除了上面介绍的，scaffold 还为我们添加了测试文件 `test/models/product_test.rb` 和 `test/controllers/products_controller_test.rb` 。

这里，Rails 默认使用的是 minitest，更多介绍可以看这里。我们也可以使用其他的测试框架，比如 Rspec。

我们可以修改 Gemfile

```
group :development, :test do
  gem 'rspec-rails'
end
```

运行 rspec 的 generator：

```
% rails generate rspec:install
create  .rspec
create  spec
create  spec/spec_helper.rb
create  spec/rails_helper.rb
```

我们补上 Model 和 Controller 的测试文件：

```
rails generate rspec:model product
rails generate rspec:controller products
```

最后，我们在 Rails 项目文件夹中运行这个命令：

```
% rspec
**

Pending: (Failures listed here are expected and do not affect your suite's status)

  1) ProductsController
     # Not yet implemented
     # ./spec/controllers/products_controller_spec.rb:4

  2) Product add some examples to (or delete) /Users/liwei/Desktop/Rails_practice_p1_0
/code/chapter_2/shop/spec/models/product_spec.rb
     # Not yet implemented
     # ./spec/models/product_spec.rb:4


Finished in 0.00058 seconds (files took 1.6 seconds to load)
```

我们看到测试文件已经可以运行了，虽然我们还未给它写一行测试用例（Test Case）。

在后面 MVC 开发的部分，我们会继续添加它的代码。Rspec 的代码和文档在这里：

https://github.com/rspec/rspec

让 Rspec 集成到 Rails 中的方法是安装 `rspec-rails`：

```
group :development, :test do
  gem 'rspec-rails', '~> 3.0'
end
```

https://github.com/rspec/rspec-rails

下一节，我们将深入 Rails 中，了解它的核心概念：REST。

2.2 REST 架构

概要：

本课时结合 Rails 路由（routes），详解 Rails 如何实现 REST 架构。

知识点：

1. REST
2. CRUD

正文

2.2.1 什么是 REST

REST，Representational State Transfer， 更准确地表述应该是：具有代表性的状态转移。这是一种软件架构风格。说它是风格，表明它不具备约束。你可以破坏它，不按照它的风格去实现。但是，REST 拥有简洁的设计理念，按照它的设计可以在开发中获得益处。

Rails 是按照 REST 风格设计的，从1.2版本起，Rails 就开始按照 REST 架构管理资源。

如何管理呢？Rails 从以下三个方面对资源进行定义：

1. 直观简短的资源地址：URI，比如：http://example.com/resources/。
2. 可传输的资源：Web 服务接受与返回的互联网媒体类型，比如：JSON，XML ，YAML 等。
3. 对资源的操作：Web 服务在该资源上所支持的一系列请求方法（比如：POST，GET，PUT或DELETE）。

在我们的代码里，我想你已经在上一节创建的项目里体验到了如何增加，修改，和删除一个商品。

以上引述于http://zh.wikipedia.org/wiki/REST，并结合 Rails 做了解释。这里还有一篇文章，推荐阅读：如何获取（GET）一杯咖啡——星巴克REST案例分析

REST是资源管理的模式，和 SOAP 和 XML-RPC 相比更加简洁，下一节，我们介绍Rails 是如何管理资源的。

2.2.2 CRUD，资源的增删改查

首先，我们在 Rails 中定义一个资源，我们在 routes 中使用 resources 这个方法：

```
Rails.application.routes.draw do
  ...
  resources :products
```

我们运行这个命令：

```
% rake routes | grep product
    products GET    /products(.:format)          products#index
             POST   /products(.:format)          products#create
 new_product GET    /products/new(.:format)      products#new
edit_product GET    /products/:id/edit(.:format) products#edit
     product GET    /products/:id(.:format)      products#show
             PATCH  /products/:id(.:format)      products#update
             PUT    /products/:id(.:format)      products#update
             DELETE /products/:id(.:format)      products#destroy
```

Rails 为我们提供了7个方法，他们在 `app/controllers/products_controller.rb` 这个文件中。

当我们 GET `/products` 这个地址时，调用的是 `index` 方法，当我们 POST `/products` 地址时，Rails 会按照 REST 的模式，把请求转入到 `create` 方法内。我们看一下这个表：

HTTP 请求方法在RESTful Web 服务中的典型应用

资源	GET	PUT	POST	DELETE
一组资源的URI，比如 http://example.com/resources/	列出URI，以及该资源组中每个资源的详细信息（后者可选）。	使用给定的一组资源替换当前整组资源。	在本组资源中创建/追加一个新的资源。该操作往往返回新资源的URL。	删除整组资源。
单个资源的URI，比如 http://example.com/resources/142	获取 指定的资源的详细信息，格式可以自选一个合适的网络媒体类型（比如：XML、JSON等）	替换/创建指定的资源。并将其追加到相应的资源组中。	把指定的资源当做一个资源组，并在其下创建/追加一个新的元素，使其隶属于当前资源。	删除指定的元素。

所以，向 `PATCH` 或 `PUT` 的地址是一个具体的资源，比如 `/products/1`，而 Rails 会把请求转移到 `update` 方法中。值得注意的是：在之前的 Rails 版本中，用的是 `PUT` 动作，4.0 后引入 `PATCH`，稍微不同的是，patch 可以表示更新或局部更新，但在使用上，和 `PUT` 无异。[1]

`:format` 表示我们可以接受和响应对应的 format 请求。比如 `/products/1` 响应的是 html，而 `/products/1.json` 响应的是 json。

我们可以关闭这种响应，只需要 `resources :products, format: false`。

或者更改响应，只接受和响应 json，如：`resources :products, format: 'json'`。

在实践中，这对 API 的设计非常方便，比如页面上 ajax 调用 `/api/users/1/status`，约束它只返回 json 格式。

从现在开始，我们的代码主要集中在 app 文件夹内。下一节，我们将深入 Rails 的 routes 中，看看实践中经常遇到的情况，以及如何解决。你也可以请先阅读以下 Rails 手册 中的 routes 章节。

阅读

REST服务开发实战

为啥REST如此重要？

2.3 深入路由（routes）

概要：

本课时详细解读如何设置复杂情况下的路由（routes），以及路由文件中常用方法。

知识点：

1. routes 定义
2. 嵌套（nested）
3. namespace
4. concern
5. 参数
6. 测试

正文

2.3.1 定义路由（routes）

上一节，我们讲了 Rails 通过 routes，来实现 REST 风格的架构。本节我们讲详细介绍下如何使用 routes，定义我们想要的地址（URL）。

我们先为项目，创建一个 controller：

```
rails g controller home index welcome about contact
```

在我们专门讲解 controller 前，先简单解释下：

- g 是 generate 的缩写，我想你已经在 2.1.1 里看到了。
- controller，说明我们创建的是一个 controller，也可以是 model。
- home 是 controller 的名字。
- index... 和其他几个名字，是 controller 中的方法，并且会自动创建对应的 views 文件。

好了，我们在它上面做一些简单的例子，打开routes，你可以看到它已经增加了几个定义：

```
get 'home/index'
get 'home/welcome'
get 'home/about'
get 'home/contact'
```

我们访问 `http://localhost:3000/home/index` 可以看到它。但是，如果我想访问 `http://localhost:3000/` 就进入到 index 方法呢？

```
get '/', to: 'home#index'
get '/welcome', to: 'home#welcome'
```

如上，我们自己定义了访问和方法之间的对应关系。其实我们更经常使用 root 来定义地址：

```
root 'home#index'
```

运行 `rake routes` ，我们可以看到

Prefix	Verb	URI Pattern	Controller#Action
home_contact	GET	/home/contact(.:format)	home#contact
	GET	/	home#index
welcome	GET	/welcome(.:format)	home#welcome
root	GET	/	home#index

我们也可以用其他的 Verb 来定义非 GET 请求，比如

```
put '/haha', to: 'home#index'
delete '/hehe', to: 'home#index'
patch '/wawa', to: 'home#index'
```

routes 中我们可以抛开资源的要求（非 REST 风格），直接设定一个访问地址：

```
get '/something/:controller/:name/:action'
```

这时我们访问 `http://localhost:3000/something/home/aaa/index` 也会进入到 `'home#index'` 中，因为 Rails 会这样解析：

- something 是个前缀
- 访问的 controller 是 home
- name 参数是 aaa
- 方法是 index

建议你看一下的终端：

```
Started GET "/something/home/aaa/index" for ::1 at 2015-02-19 17:10:26 +0800
Processing by HomeController#index as HTML
  Parameters: {"name"=>"aaa"}
```

Rails 已经将你的请求转移到对应的 controller 中了。

如果一个地址，即可以接收 post 请求，也可以接收 get 等请求，我们可以使用 match 方法：

```
match ':controller/:action/:id', via: [:get, :post]
```

提示：在开发（development）环境中，修改 routes 是不需要重启服务的。

2.3.1.1 扩展 resources

前面我们已经定义了一个 `resource :products` ，这在实际开发中还是不够的，比如，一个 Product 下如果查看评论，比如，显示卖的最好的十个 Products：

```
resources :products do
  collection do
    get :top # 排行榜功能
  end
  member do
    post :buy # 添加到购物车
  end
end
```

运行 `rake routes` 可以看到：

Prefix	Verb	URI Pattern	Controller#Action
top_products	GET	/products/top(.:format)	products#top
buy_product	POST	/products/:id/buy(.:format)	products#buy

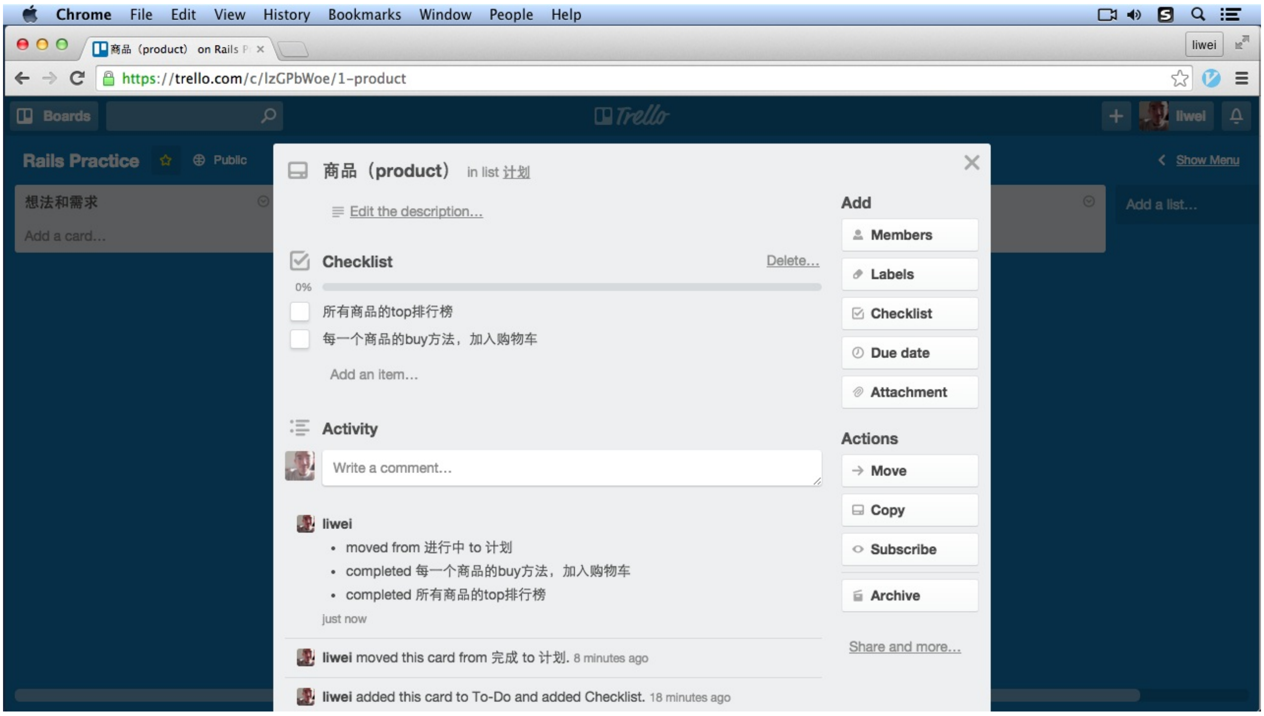
不同的是，collection 用于 products 中增加方法，member 给具体一个 product 增加方法。

补充一点，我们可以在一行里，定义多个 resources，比如：

```
resources :photos, :books, :videos
```

虽然方便，但不够灵活，实践中还是要按照需求调整的。

我们在这里提出了两个功能需求：top 排行榜，和添加到购物车。这里我使用 trello.com 来记录这两个需求。



我在“计划”里增加了一个 card，在 checklist 中记录了这两个需求。当我们开始功能开发的时候，可以将 card 拖动到“进行中”，当我们完成一个功能的时候，可以在 checklist 的项目前打一个√，当我们完成一个 card 的任务后，可以讲 card 拖动到“完成”中。

Rails 被很多开发团队使用，在一些开发团队中，经常会提到敏捷开发，trello 是一个很好的敏捷开发工具，可以方便的管理我们的日常工作，和记录项目进展状态。

2.3.1.2 单个资源 resource

```
resource :settings
resource :profile
```

这是设定一个单数资源的方法，项目里，哪些是单数呢？比如系统设定，比如当前用户的个人信息，运行 `rake routes` 可以看到，它是没有 `:id` 这个参数的。

在这个例子里，我们还未给 settings 和 profile 创建 controller 和 view，不过这不妨碍 routes 产生我们想要的地址。

2.3.1.3 选择方法

`resources` 给我们创建了七个方法，但是不见得我们都要用到，为了代码的整洁[1]，我们可以做一些排除：

```
resources :users, only: [:index, :show]
resources :products, except: [:destroy]
```

`only` 表示我们需要的方法，`except` 表示我们不需要的方法。通常，我们的确会像上面这么做，比如我们的网站只提供用户（User）的列表和查看功能，而管理功能（增删改）要在管理界面进行，而它的地址一般不会是 `/users/1/edit` 这样，而是 `/admin/users/1/edit` 。

[1]这是个人癖好，有的人的确不愿意这么做，不过 Rails 给了我们让项目变得“整洁”的方法。

2.3.1.5 地址解析的辅助方法

刚才，我们讲到了 `_path` 这个后缀，Rails 还有一个 `_url` 。

地址	结果
products_path	'/products'
products_url	'http://localhost:3000/products'

`_path` 和 `_url` 是 routes 的辅助方法，我们在下一章将详细介绍。

2.3.2 嵌套的路由（routes）

在我们定义资源的时候，有时候一个资源会有它的子资源，比如一个商品（product）会有多个商品种类（variants），当我们购买一个商品的时候，也需要选择哪个种类，比如T恤的种类氛围尺码，而每一个尺码有不同的价格。

这时该如何定义 routes 呢？

```
resources :products do
  resources :variants
end
```

运行 `rake routes` ，可以看到一个商品（product）下，增加了这些routes：

Prefix	Verb	URI Pattern	Co
product_variants	GET	/products/:product_id/variants(.:format)	var
	POST	/products/:product_id/variants(.:format)	var
new_product_variant	GET	/products/:product_id/variants/new(.:format)	var
edit_product_variant	GET	/products/:product_id/variants/:id/edit(.:format)	var
product_variant	GET	/products/:product_id/variants/:id(.:format)	var
	PATCH	/products/:product_id/variants/:id(.:format)	var
	PUT	/products/:product_id/variants/:id(.:format)	var
	DELETE	/products/:product_id/variants/:id(.:format)	var

我们为 variants 也使用一下 `scaffold` ：

```
rails g scaffold variant product_id:integer price:decimal size
```

在运行 `rails s` 前，记得要更新数据库：

```
rake db:migrate
```

记得，我们应该删除 routes 中自动添加的 `resources :variants` ，因为我们不需要在 `http://localhost:3000/variants` 下看到它，不是么？我们可以在每一个商品（Product）页面，比如： `http://localhost:3000/products/1` 中看到它了。

2.3.3 路由中的命名空间（namespace）

接下来我们说两个项目中经常会见到的情形。

一个项目，肯定要有 admin 的，我们如何把管理地址都放到 http://localhost:3000/admin/ 这个目录下？

```
namespace :admin do
  resources :products
end
```

这时，这样就足够了，不过，它所使用的 controller 和 view 是在 admin 这个文件夹下面的，多说一点，它的controller 代码也是在 Admin 这个 module 下的。如果你还对 Ruby 的 module 不熟悉，是时候补充下了。

它的代码是：

```
class Admin::ProductsController < ApplicationController
  ...
end
```

这里，我们反过来想，能否让 `/admin/articles` 下的代码去访问 ArticlesController？这里不再是 `Admin::` 开头的。这时我们用到 `scope` ：

```
scope '/admin' do
  resources :articles
end
```

对于 admin 下的资源管理，可以试试 active admin 这个 Gem。

https://github.com/activeadmin/activeadmin

2.3.4 concern 方法

再来看一个让 routes 更简洁，也很实用的方法。

```
concern :commentable do
  resources :comments
end

concern :image_attachable do
  resources :images, only: :index
end

resources :messages, concerns: :commentable
resources :articles, concerns: [:commentable, :image_attachable]
```

`concern` 定义好的资源，可以被其他 resource 里多次引用。

Rails 的原则之一：`不要重复自己（Don't Repeat Yourself）`

2.3.5 有用的参数

:as 别名

如果再上面地址后面，加上 `as` 参数，会直接创建一个别名的地址，比如

```
get 'home/welcome', as: :welcome
```

之前，我们在 views 或者 controller 中，连接到或跳转到 `/home/index` 可以这么写：`home_welcome_path`，增加了 `:as` 后，就变成了 `welcome_path` 了。好处是，如果我们某一天更改了对应的 action 甚至 controller，这个写法 `welcome_path` 是不会变的，而只需要改动 routes 中的定义。

在定义 routes 时，要注意不要重复定义，因为：写在上面的会覆盖下面的。比如：

```
get 'home/index', to: 'home#welcome'
get 'home/index'
```

访问 `http://localhost:3000/home/index` 会进入到 `welcome` 方法中。

下面在介绍几个实用的参数。

shallow

这时 Rails 4 中增加的一个很实用的参数。

```
resources :products do
  resources :comments, shallow: true
end
```

它把 index、new 和 create 方法保留在了 `products/:id` 这个资源下，而把其他方法，重新放回到 `/comments` 下。这样的考虑是避免过多的实用嵌套 routes，并且让代码更简洁。

constraints

我们可以给 routes 建立约束（Constraints），比如：

```
get 'photos/:id', to: 'photos#show', constraints: { id: /[A-Z]\d{5}/ }
```

这时，id 为 A 到 Z 开头，且后面为5位数字的 id，才符合路由条件，转入到 `show` 方法。而 `products/A123456` 将会提示 `No route matches`。

2.3.6 Rspec 测试

通常，我们会在 controller 中写上测试，不过 Rspec 也为我们提供了测试路由的方法。我们在 spec 下建立一个routing 文件夹，并且添加一个 `products_routing_spec.rb` 的文件：

```
RSpec.describe ProductsController, type: :routing do
  describe "routing" do

    it "routes to #index" do
      expect(:get => "/products").to route_to("products#index")
    end
    ...
```

我们为它单独运行测试，因为scaffold 自动为我们添加的测试代码，我们将在后面的章节完成：

```
% rspec spec/routing/products_routing_spec.rb
```

routes 测试的参考，可以查看这里。

好了，本章结束了，本节的内容多来自 Rails 手册中的 Rails Routing from the Outside In，你也可以找到在 这里 找到本章调试的代码。下一章，我们将开始完成 shop 的页面（views）代码，希望它可以让你更加了解 Rails。

第三章 Rails 中的视图

课程概要：

本课程讲解Rails 视图（View），内容包括常用的辅助方法（Helper），如何使用表单（Form），AJAX在视图中的应用以及如何借助其他的模板引擎实现简洁的页面方案。

知识点：

1. 布局
2. 辅助方法
3. 表单
4. AJAX
5. 模板引擎

课程背景

视图（View）即 MVC 中的 V，也是Rails使用者最先见到的部分。在完成业务逻辑前，合理的设计视图是 MVC 开发中最先得到用户认可的部分。本课程结合商品页面的开发，讲解 Rails 中的视图。

3.1 布局和辅助方法

概要：

本课时讲解Rails 视图（View）中的布局文件，常见的辅助方法（Helper）以及如何使用局部模板。

知识点：

1. 布局（layout）
2. 辅助方法（helper）
3. 局部模板（partial）

正文

3.1.1 布局（layout）

本章开始，我们将进入 Rails 的视图（view） 的开发中。如果你对 Rails 这个 MVC 框架还有一些模糊的话，建议读一读这篇文章。

> Rails 是一个 RESTful 风格的 MVC 框架。

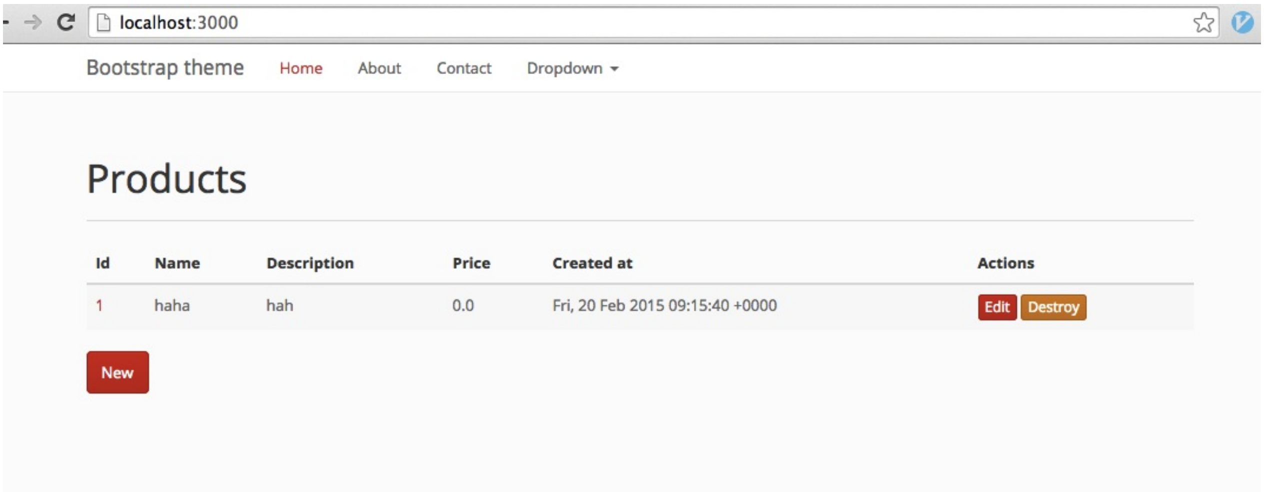
我们把第一章使用 bootswatch 创建的项目 copy 过来。现在，我们进入到 `app/views` 这个文件夹吧。

`layouts` 里放的是布局文件。如果我们网站只有一种布局，那么一个 `application.html.erb` 就足够了。我们也可以为每个资源创建一个 `layout` ，比如 `app/views/layouts/products.html.erb` 。

我们删掉多余的代码，增加一个 `yield` 的辅助方法（helper）。

```
<div class="container">
  <%= yield %>
</div>
```

访问我们的页面，希望你会看到和我一样的效果。如果没有，没关系，可以到 这里 clone 我们的代码。



`yield` 方法可以让 Rails 使用我们的模板（template） `app/views/products/index.html.erb` 填充了布局（layout）。

我们再看一下 `app/views/layouts/application.html.erb` 中的这一行：

```
<%= yield(:page_stylesheet) if content_for?(:page_stylesheet) %>
```

如果我们在 `app/views/products/index.html.erb` 中使用 `content_for` 方法，可以在这个 layout 的这个位置，显示额外的内容，比如，我们在 `index.html.erb` 的最上面增加：

```
<%= content_for :page_stylesheet do %>
<!-- 这是 index.html.erb 里单独使用的 -->
<% end %>
```

再刷新下页面，我们在源码里看到：



在实践开发里，我们经常这样做：布局中加载的是所有页面通用的内容和 css，js，而到了具体页面，就通过 `content_for` 这个辅助方法定义自己的内容，在我们的 `application.html.erb` 里，你可以找到四个 `content_for`，这样给我们的代码里增加了一些灵活，也不必把所有内容都写到一起。

`content_for?` 判断我们是否定义了这个变量。

如果我们想更改一下布局，该如何做呢？实践中，我们的确会遇到以下几种情形：

情形一：admin 要使用自己的布局文件 `app/views/layouts/admin.html.erb`

我们在 admin 的 controller 里声明它使用另一个：

```
class AdminController < ApplicationController
  layout "admin"
```

通常我们把 admin 放到 module 中，而为 admin 建立一个通用的 controller，让所有 admin 的 controller 都继承它，这样，我们不用反复的去定义了：

```
class Admin::BaseController < ApplicationController
  layout "admin"
end

class Admin::ProductsController < Admin::BaseController
end

class Admin::CommentsController < Admin::BaseController
end
```

情形二：为完成某个特殊操作，我们需要更改布局。

这时，我们要在 action 里去变更这个布局，比如，创建一个 Product 的时候：

```
def new
  @product = Product.new
  render layout: "another_layout"
end
```

情形三：不用布局

```
def edit
  render layout: false
end
```

3.1.2 常用的辅助方法（helper）

上一节，我们已经使用了几个辅助方法，这里我们再介绍几个 Layouts and Rendering in Rails 提到的 helper。

link_to

你会发现在页面里最多的是 link_to 这个方法，它的参数也是蛮多的，我们来详细讲解。

我们把现在的 view 修改一下，把 `首页` 的链接加上。

```
<%= link_to "网店演示", root_path, class: "navbar-brand" %>
```

一个稍复杂的例子：

```
<%= link_to "删除", product, :method => :delete, :data => { :confirm => "点击确定继续" },
 :class => 'btn btn-danger btn-xs' %>
```

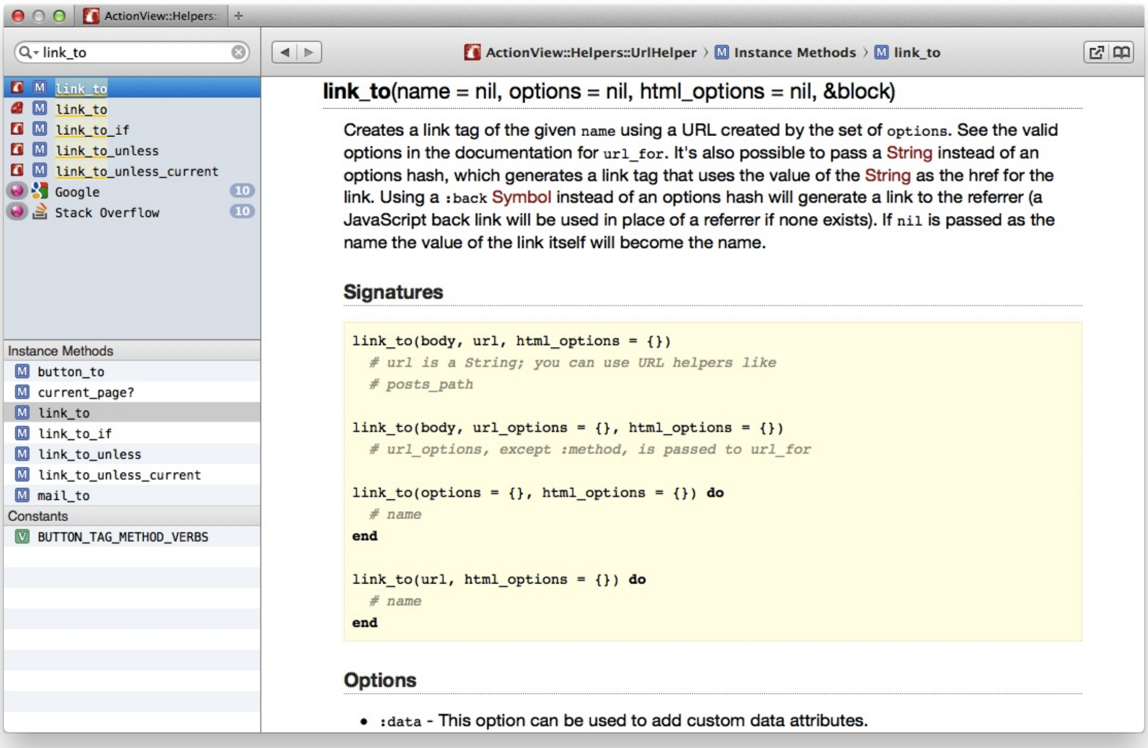
我们可以改变 `link_to` 的默认行为（GET），`:method => :delete` 将发送 `delete` 请求。`:confirm` 将会告诉浏览器阻止我们当前的动作，直到点击 `确定` 。实现上面两个效果，需要引入 `ujs`，在我们的 `app/assets/javascripts/simplex.js` [1] 中已经为我们引入了：

```
//= require jquery
//= require jquery_ujs
```

注[1]：通常我们使用的是 `application.js` ，但是在 1.3 中我们设计了新的主题，目前我使用的是 `simplex` 。

写到这里，我要推荐 http://api.rubyonrails.org/ 这里了，对于各种 Rails 本身的方法，我们可以通过查询 api 文档得到。如果是某个 Gem 提供的方法，我们可以直接翻看它的 README 或者代码。

一个较实用的工具 Dash，可以帮你管理每个版本 api 文档，查询起来也很方便。不过它是收费的。



image_tag

建议你使用 api 文档查找一下这方法，你会看到这个代码提示：

```
image_tag("icon")
# => <img alt="Icon" src="/assets/icon" />
image_tag("icon.png")
# => <img alt="Icon" src="/assets/icon.png" />
image_tag("icon.png", size: "16x10", alt: "Edit Entry")
# => <img src="/assets/icon.png" width="16" height="10" alt="Edit Entry" />
image_tag("/icons/icon.gif", size: "16")
# => <img src="/icons/icon.gif" width="16" height="16" alt="Icon" />
image_tag("/icons/icon.gif", height: '32', width: '32')
# => <img alt="Icon" height="32" src="/icons/icon.gif" width="32" />
image_tag("/icons/icon.gif", class: "menu_icon")
# => <img alt="Icon" class="menu_icon" src="/icons/icon.gif" />
```

我们的图片是来自 `app/assets/images` 的，我放了一个 `logo.png` 在里面，你会发现它的地址是： `http://localhost:3000/assets/logo-be2e3e66a18126c4042f84cd4aae4cb3.png` 。Rails 使用 sprockets-rails 来管理 `app/assets` 中的文件，后面章节我们会详细介绍。

这里，我们可以关闭 `be2e3e66a18126c4042f84cd4aae4cb3` 这种形式：

`config/environments/development.rb`

```
config.assets.digest = false
```

重启我们的服务，地址变为 `http://localhost:3000/assets/logo.png` 。

auto_discovery_link_tag

我们经常在 `head` 里和页面里，增加 rss 和 atom 订阅连接，这时，我们可以使用 `auto_discovery_link_tag` 这个辅助方法。

```
<head>
...
<%= auto_discovery_link_tag(:rss, {controller: "products", action: "index"}, {title: "
RSS Feed"}) %>
<%= auto_discovery_link_tag(:atom, {controller: "products", action: "index"}, {title:
"ATOM Feed"}) %>
...
</head>
```

我们也可以在页面中增加这个连接，这在 web2.0 兴起后的博客中很常见，方便我们把数据加入到订阅中。

```
<%= link_to "rss", products_url(format: "rss") %>
<%= link_to "atom", products_url(format: "atom") %>
```

剩下的问题是，Rails 如何提供这个数据，我并不想等到 controller 里再去讲这个部分，让我们现在开始了解下：

> Rails 是会根据我们的请求类型，做出响应。

如果我们请求的是一个 `http://localhost:3000/products.html`，Rails 会给我们 html 的页面，而如果我们请求的是 `http://localhost:3000/products.rss`，Rails 会自动选择 rss 的模板，渲染（render）后返回我们结果。`http://localhost:3000/products.atom` 也是一样。所以，我们在 `app/views/products/` 中增加两个文件：`index.rss.builder` 和 `index.atom.builder`。

在 controller 里，如果我们想对结果做一些其他的操作，就需要增加这个代码：

`app/controllers/products_controller.rb`

```
respond_to do |format|
  format.html
  format.rss { ... }
  format.atom { ... }
end
```

在这个例子中，我们并不需要改变什么，所以不用添加它。

`app/views/products/index.atom.builder`

```
atom_feed do |feed|
  feed.title "商品列表"
  feed.updated @products.maximum(:updated_at)

  @products.each do |product|
    feed.entry product, published: product.updated_at do |entry|
      entry.title product.name
      entry.content product.description
      entry.price product.price
    end
  end
end
```

`app/views/products/index.rss.builder`

```
xml.instruct! :xml, version: "1.0"
xml.rss version: "2.0" do
  xml.channel do
    xml.title "商品列表"
    xml.description "这是商品列表"
    xml.link products_url

    @products.each do |product|
      xml.item do
        xml.title product.name
        xml.description product.description
        xml.price product.price
        xml.link product_url(product)
        xml.guid product_url(product)
      end
    end
  end
end
```

再次访问 `http://localhost:3000/products.rss` 和 `http://localhost:3000/products.atom` ，你会发现我们得到了结果。

我们用到了 `.builder` 这个结尾的文件，它会告诉 Rails 使用 `Builder::XmlMarkup` 这个库（lib）来解析文件。所以看 `rss.builder` ，它是按照 xml 格式写的。 `atom.builder` 用到了另一个辅助方法 `atom_feed` ，写法虽然不同，但是生成的内容也是 xml 格式的。

在 `scaffold` 创建的文件里，你会看到 `index.json.jbuilder` ，它会使用 `JBuilder` 这个库来解析并生成 `json` 的结果。这会在后面的章节讲到，你可以在 这里 先了解一下。

Railscasts.com 是所有 Rails 学习者必看的网站，这个 视频 一定会帮助你理解上面的内容。

在此，向 Ryan 致敬。

stylesheet_link_tag

```
<head>
<%= stylesheet_link_tag "simplex", :media => "all" %>
</head>
```

css 文件的引用通常放到页面的 `head` 标签之间。这里我们引用的是 css 文件，我们也可以把它改为 `.css.scss` ，这样可以在里面写一些 `scss` 语法，而不用更改我们的引用。我们在 `2.1.3` 中已经提到了 `scss` 。

javascript_include_tag

```
...
<%= javascript_include_tag "simplex" %>
<%= yield(:page_javascript) if content_for?(:page_javascript) %>
</body>
```

浏览器是自上而下解析节点元素（DOM）的，所以，请注意我们把 `js` 文件加载放到页面最下面，以免因为某个 js 解析问题导致页面始终无法显示。在引用完 `js` 库后，我们还可以根据需要，单独放置页面的 `:page_javascript` 。

ActionView 还为我们提供了其他很多辅助方法，可以查看 这里。

3.1.3 局部模板（partial）

> DRY, Don't Repeat Yourself.不要重复自己。

为了让我们节省更多的页面重复代码，我们还可以使用局部模板（partial）。打开我们的 `app/views/products/index.html.erb` ：

```
<% @products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>
```

这里我们使用了局部模板， `partial` 指定了使用哪个模板， `locals` 向模板里传递了一个变量。在 `_product.html.erb` 里，我们显示具体 `product` 的信息。

不过这是一个老套的写法，Rails 4 给了我们更酷的写法：

```
<%= render @products %>
```

不过，如果需要传递更多的变量（locals），还是要用第一种方法，当然，你完全可以把 `each do` 的代码放到局部模板里。

我们也可以不传递变量到局部模板里，它可以找到 `@products` ，看一下 `new.html.erb` 和 `edit.html.erb` ：

```
<%= render :partial => 'form' %>
```

也可以直接写：

```
<%= render 'form' %>
```

如果我们在页面加载路径中，放置了多个同名的局部模板，它会显示离它最近的那个。我们可以把公用较多的模板，放到一个专属的文件夹里，比如 `shared` ，引用的时候：

```
<%= render partial: "shared/product", locals: { product: product } %>
```

注：当使用 `locals` 传递参数时，一定要声明 `partial` 。

我为大家在 `shared` 中放置了一个同样的 `_product.html.erb` ，大家可以在 `index.html.erb` 中调用看看。

说一个实践中经常用到的局部模板。

我建立了一个新的文件夹 `application` ，这里会放布局文件使用的局部模板，我放了一个 `_flash.html.erb` ，这是 `flash` 通知。看看我们的 `products_controller.rb` ，我们在操作成功后会有提示信息，它在我们的页面上还没有显示。我修改了一下 `application.html.erb` ：

```
<div class="container">
  <%= render "flash" %>
  <%= yield %>
  ...
```

为了让 `flash` 符合 `bootstrap` 的格式，我做了代码调整，大家可以参考代码。 `flash` 是 session 的应用，通常在 controller 的 `action` 间传递信息，读取成功后自动清空。如果一个 flash 没有在合适得地方读出来，那么它将被保存到读出为止，这会造成本不该显示它的地方却显示它，所以我把 `flash` 放到了 `layout` 中，使得所有页面都会引用它，保证它产生后立刻显示，并在显示后自动清空。

3.2 表单

概要：

本课时讲解 Rails 如何通过表单（Form）传递数据，以及表单中的辅助方法使用，并实现登陆注册功能。

知识点：

1. 表单
2. 表单中的辅助方法（helper）
3. 表单绑定模型（Model）
4. 注册和登录

正文

3.2.1 搜索表单（Form）

如果我们的表单不产生某个资源的状态改变，我们可以使用 GET 发送表单，这么说很抽象，比如一个搜索表单，就可以是 GET 表单。

我们在页面的导航栏上，增加一个搜索框：

```
<%= form_tag(products_path, method: "get") do %>
  <%= label_tag(:q) %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("搜索") %>
<% end %>
```

`form_tag` 产生了一个表单，我们设定它的 `method` 是 `get`，它的 `action` 地址是 `products_path`，我们也可以设定一个 hash 来制定地址，比如：

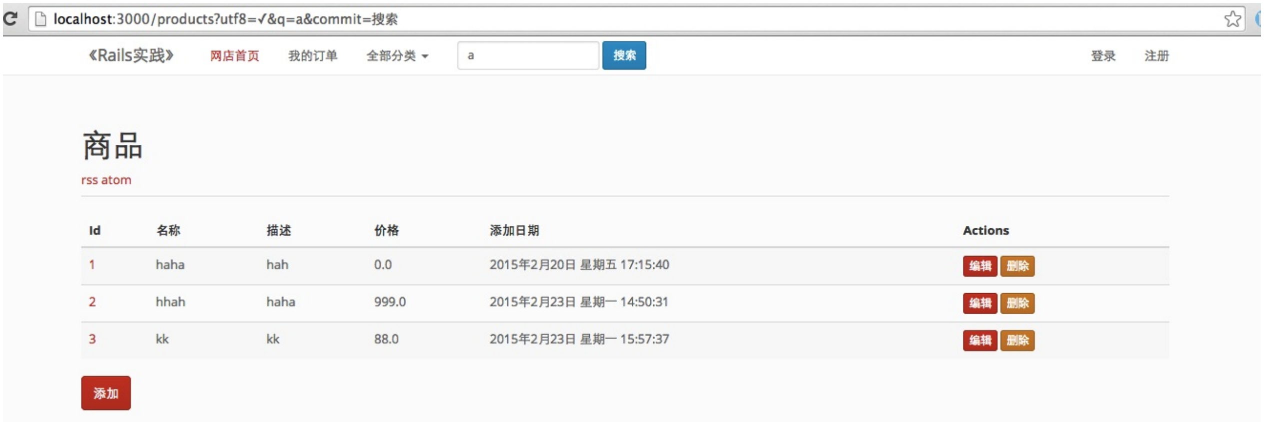
```
form_tag({action: "search"}, method: "get") do
```

这需要你在 `products` 里再增加一个 `search` 方法，否则，你会得到一个 `No route matches {:action=>"search", :controller=>"products"}` 的提示，这告诉我们，`form_tag` 的第一个参数需要是一个可解析的 `routes` 地址。当然，你也可以给它一个字符串，这个地址即便不存

在，也不会造成 `no route` 提示了。

```
form_tag("/i/dont/know", method: "get") do
```

这并不是我们最终的代码，我们还需要增加一些附加的属性，让我们的式样看起来正常一些。而且我用了 `params[:q]` 这个方法，获得地址中的 `q` 参数，把搜索的内容放回到搜索框中。



我们可以在 controller 里，使用 ActiveRecord 的 where 方法查询传入的参数，我们也可以使用 (ransack)[https://github.com/activerecord-hackery/ransack] 这种 gem 来实现搜索功能。

ransack 是一个 metasearch 的 gem，实现它非常的方便。我们把它加入到 gemfile 里：

```
gem 'ransack'
```

我们在视图里，使用 ransack 提供的辅助方法，来实现表单:

```
<%= search_form_for @q, html: { class: "navbar-form navbar-left" } do |f| %>
  <div class="form-group">
    <%= f.search_field :name_cont, class: "form-control", placeholder: "输入商品名称" %>
  </div>
<% end %>
```

提示：如果每个页面都包含这个搜索框，但是不见得每个页面都有 @q 这个实例，所以我们可以自己写一个表单，实现搜索：

```
<%= form_tag products_path, method: :get, class: "navbar-form navbar-left" do %>
  <div class="form-group">
    <%= text_field_tag "q[name_cont]", params["q"] && params["q"]["name_cont"], class:
 "form-control input-sm search-form", placeholder: "输入商品名称" %>
  </div>
<% end %>
```

在商品的 controller 中，我们修改 index 方法：

```
def index
  @q = Product.ransack(params[:q])
  @products = @q.result(distinct: true)
end
```

好了，一个简单的查询实现了。这里我们使用的是 `name_cont` 来实现模糊查询，文档 上提供了详尽的方法，实现更复杂的查询。

3.2.2 常用的表单辅助方法

在我们使用 `form_tag` 的同时，我们还需要一些辅助方法来生成表单控件。

```
<%= text_area_tag(:message, "Hi, nice site", size: "24x6") %>
<%= password_field_tag(:password) %>
<%= hidden_field_tag(:parent_id, "5") %>
<%= search_field(:user, :name) %>
<%= telephone_field(:user, :phone) %>
<%= date_field(:user, :born_on) %>
<%= datetime_field(:user, :meeting_time) %>
<%= datetime_local_field(:user, :graduation_day) %>
<%= month_field(:user, :birthday_month) %>
<%= week_field(:user, :birthday_week) %>
<%= url_field(:user, :homepage) %>
<%= email_field(:user, :address) %>
<%= color_field(:user, :favorite_color) %>
<%= time_field(:task, :started_at) %>
<%= number_field(:product, :price, in: 1.0..20.0, step: 0.5) %>
<%= range_field(:product, :discount, in: 1..100) %>
```

解析后的代码是：

```
<textarea id="message" name="message" cols="24" rows="6">Hi, nice site</textarea>
<input id="password" name="password" type="password" />
<input id="parent_id" name="parent_id" type="hidden" value="5" />
<input id="user_name" name="user[name]" type="search" />
<input id="user_phone" name="user[phone]" type="tel" />
<input id="user_born_on" name="user[born_on]" type="date" />
<input id="user_meeting_time" name="user[meeting_time]" type="datetime" />
<input id="user_graduation_day" name="user[graduation_day]" type="datetime-local" />
<input id="user_birthday_month" name="user[birthday_month]" type="month" />
<input id="user_birthday_week" name="user[birthday_week]" type="week" />
<input id="user_homepage" name="user[homepage]" type="url" />
<input id="user_address" name="user[address]" type="email" />
<input id="user_favorite_color" name="user[favorite_color]" type="color" value="#00000
0" />
<input id="task_started_at" name="task[started_at]" type="time" />
<input id="product_price" max="20.0" min="1.0" name="product[price]" step="0.5" type="
number" />
<input id="product_discount" max="100" min="1" name="product[discount]" type="range" /
>
```

更多的表单辅助方法，建议大家直接查看这个部分的 源代码，我一直认为源代码是最好的教材。

3.2.3 模型（Model）的辅助方法

我们还可以使用不带有 `_tag` 结尾的辅助方法，来显示一个模型（Model） 实例（Instance），比如我们的 `@product`，可以在它的编辑页面中这样来写：

```
<%= text_field(:product, :name) %>
```

他会给我们

```
<input type="text" value="测试商品" name="product[name]" id="product_name">
```

它接受两个参数，并把它拼装成 `product[name]`，并且把 `value` 赋予这个属性的值。我们提交表单的时候，Rails 会把它解释成 `product: {name: '测试商品', ...}`，这样，`Product.create(...)` 可以添加这个商品信息到数据库中了。

不过这样做会有个问题，这个商品会有很多属性需要我们填写，会让代码变得“啰嗦”。这时，我们可以把这个实例，绑定到表单上。

注：说模型对象，通常指 `Product` 这个模型，说模型实例，指 `@product`。一些文档上并不区分这种称呼，个人觉得容易混淆。

3.2.4 把模型（Model）绑定到表单上

来看看我们的商品添加界面使用的表单吧，它在这里 `app/views/products/_form.html.erb`

```
<%= form_for @product, :html => { :class => 'form-horizontal' } do |f| %>
```

这里我们用了 `form_for` 这个方法，它可以将一个资源和表单绑定，这里我们将 `controller` 中的 @product 和它绑定。`form_for` 会判断 @product 是否为一个新的实例（你可以看看 `@product.new_record?` ），从而将 `form` 的地址指向 `create` 还是 `update` 方法，这是符合我们之前提到的 REST 风格。

当然，大多数浏览器是不支持 `PUT` ，`PATCH` ，`DELETE` 方法的，浏览器在提交表单时，只会是 `GET` 或 `POST` ，这时，form_tag 会创建一个隐藏空间，来告诉 Rails 这是一个什么动作。而 `form_for` 会根据实例，来自动判断。

```
<input name="_method" type="hidden" value="patch" />
```

在我们显示商品属性的时候，用到了 `f.text_field :name` 这个辅助方法，这样，我们不用再为每一个 `text_field` 去声明这是哪个实例了。`f` 是一个表单构造器（Form Builder）实例，你可以在 这里 看到更多它的介绍。

我们可以自己定义 `FormBuilder` ，以节省更多的代码，也可以使用 simple form，formtastic 这种 Gem。推荐 ruby-toolbox.com 这个网站，你可以发现其他的好用的 Gem。

3.2.5 注册和登录

现在，我们实现一个很重要的功能，注册和登录。我们不需要从头实现它，因为我们有 Rails 十大必备 Gem 中的第一位：Devise 可以选择。

在 `Gemfile` 中增加

```
gem 'devise'
```

在 `bundle install` 之后，我们需要创建配置文件：用户（User）

```
% rails generate devise:install User
create  config/initializers/devise.rb
create  config/locales/devise.en.yml
 ===============================================================================

Some setup you must do manually if you haven't yet:

  1. Ensure you have defined default url options in your environments files. Here
     is an example of default_url_options appropriate for a development environment
     in config/environments/development.rb:

       config.action_mailer.default_url_options = { host: 'localhost', port: 3000 }

     In production, :host should be set to the actual host of your application.

  2. Ensure you have defined root_url to *something* in your config/routes.rb.
     For example:

       root to: "home#index"

  3. Ensure you have flash messages in app/views/layouts/application.html.erb.
     For example:

       <p class="notice"><%= notice %></p>
       <p class="alert"><%= alert %></p>

  4. If you are deploying on Heroku with Rails 3.2 only, you may want to set:

       config.assets.initialize_on_precompile = false

     On config/application.rb forcing your application to not access the DB
     or load models when precompiling your assets.

  5. You can copy Devise views (for customization) to your app by running:

       rails g devise:views

===============================================================================
```

之后，我们创建用户（User）模型：

```
% rails generate devise User
      invoke  active_record
      create    db/migrate/20150224071758_devise_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
      insert    app/models/user.rb
       route  devise_for :users
```

之后，我们创建用户（User）需要的 `views`：

```
% rails g devise:views
      invoke  Devise::Generators::SharedViewsGenerator
      create    app/views/users/shared
      create    app/views/users/shared/_links.html.erb
      invoke  form_for
      create    app/views/users/confirmations
      create    app/views/users/confirmations/new.html.erb
      create    app/views/users/passwords
      create    app/views/users/passwords/edit.html.erb
      create    app/views/users/passwords/new.html.erb
      create    app/views/users/registrations
      create    app/views/users/registrations/edit.html.erb
      create    app/views/users/registrations/new.html.erb
      create    app/views/users/sessions
      create    app/views/users/sessions/new.html.erb
      create    app/views/users/unlocks
      create    app/views/users/unlocks/new.html.erb
      invoke  erb
      create    app/views/users/mailer
      create    app/views/users/mailer/confirmation_instructions.html.erb
      create    app/views/users/mailer/reset_password_instructions.html.erb
      create    app/views/users/mailer/unlock_instructions.html.erb
```

最后，更新 db：

```
rake db:migrate
```

在使用注册登录功能前，我们修改一下布局页面，增加几个链接：

```
<% if user_signed_in? %>
  <li><%= link_to current_user.email, profile_path %></li>
  <li><%= link_to "退出", destroy_user_session_path, method: :delete %></li>
<% else %>
  <li><%= link_to "登录", new_user_session_path %></li>
  <li><%= link_to "注册", new_user_registration_path %></li>
<% end %>
```

现在，我们可以使用注册登录功能了，是不是很简单呢？

接下来，我们对 Devise 创建的页面做一点修改，同时看看 Rails 如何实现表单的。

我们登录界面在 `app/views/users/sessions/new.html.erb`，我们把它改一下，符合我们页面风格，具体如何使用 html 代码，可以参考 http://bootswatch.com/simplex/。

本章的代码在 这里，希望可以帮助大家理解表单和其使用。

3.3 视图中的 AJAX 交互

概要：

本课时通过对商品的添加、编辑和删除，讲解视图中如何使用 UJS，jQuery 和 JSON，实现无刷新情况下的页面更新。

知识点：

1. jQuery
2. UJS
3. AJAX
4. JSON

正文

上一节，我们讲解了 Rails 中的视图（View），我们再回顾一下这个视图是如何产生的：我们向服务器发起一个请求，服务器返给我们结果，查看源代码，它是一篇 `HTML` 的代码。

我们每次请求一个地址，都会给我们完整的 HTML 结果，对于内容较少的网页，传输起来还是很快的，但是对于内容多的网页，大篇的结果自然会拖慢页面显示。

当我们浏览页面的时候，并不期望总是刷新整个页面，因为它没必要。现在我们有 ajax 技术，可以只加载和显示部分页面代码。举个简单的例子：当我们提交了一条评论，页面上自动显示出我们提交的评论内容。我们点击购买按钮，页面上就提示我们购物车里增加了一个商品。而这些，都不必要刷新整个页面。

ajax 是 Asynchronous Javascript And XML 的缩写，含义是异步的 js 和 XML 交互技术。XML，可扩展标记语言，我们使用的 HTML 是基于其发展起来的。

下面我们看下 Rails 是如何把 ajax 技术应用在视图（View）中的。

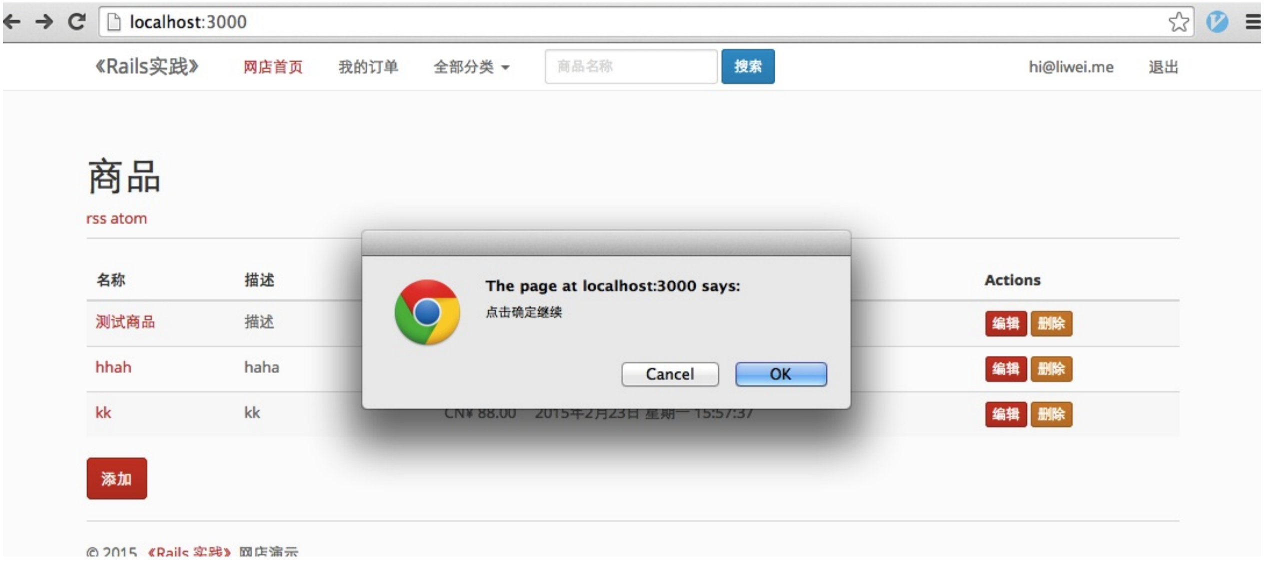
3.3.1 ujs

我们在 Gemfile 中已经使用了 `gem 'jquery-rails'` 这个 Gem，它可以让我们在 `application.js` 中增加这两行：

```
//= require jquery
//= require jquery_ujs
```

jQuery 是一个轻量级的 js 库，可以方便的处理HTML，事件（Event），动态效果，为页面提供 ajax 交互。jQuery 有很完善的文档及演示代码，以及大量的插件。

Rails 使用一种叫 ujs（Unobtrusive JavaScript）的技术，将 js 应用到 DOM 上。我们来看一个例子：



我们已经给删除连接增加了两个属性：

```
<%= link_to "删除", product, :method => :delete, :data => { :confirm => "点击确定继续" }
%>
```

来看看我们的 HTML：

```
<a data-confirm="点击确定继续" rel="nofollow" data-method="delete" href="/products/1">删
除</a>
```

辅助方法 `link_to` 使用了 `:data => { :confirm => "点击确定继续" }` 这个属性，为我们添加了 `data-confirm="点击确定继续"` 这样的 HTML 代码，之后 ujs 将它处理成一个弹出框。

在删除按钮上，还有 `:method => :delete` 属性，这为我们的连接上增加了 `data-method="delete"` 属性，这样，ujs 会把这个点击动作，会发送一个 `delete` 请求删除资源，这是符合 REST 要求的。

我们可以给 `a` 标签增加 `data-disable-with` 属性，当点击它的时候，使它禁用，并提示文字信息。这样可以防止用户多次提交表单，或者重复的链接操作。

我们为商品表单中的按钮，增加这个属性：

```
<%= f.submit nil, :data => { :"disable-with" => "请稍等..." } %>
```

当我们提交表单时，会有：



如果你还没看清楚效果，页面就已经跳转了，我们可以给 create 方法增加一个 `sleep 10`：

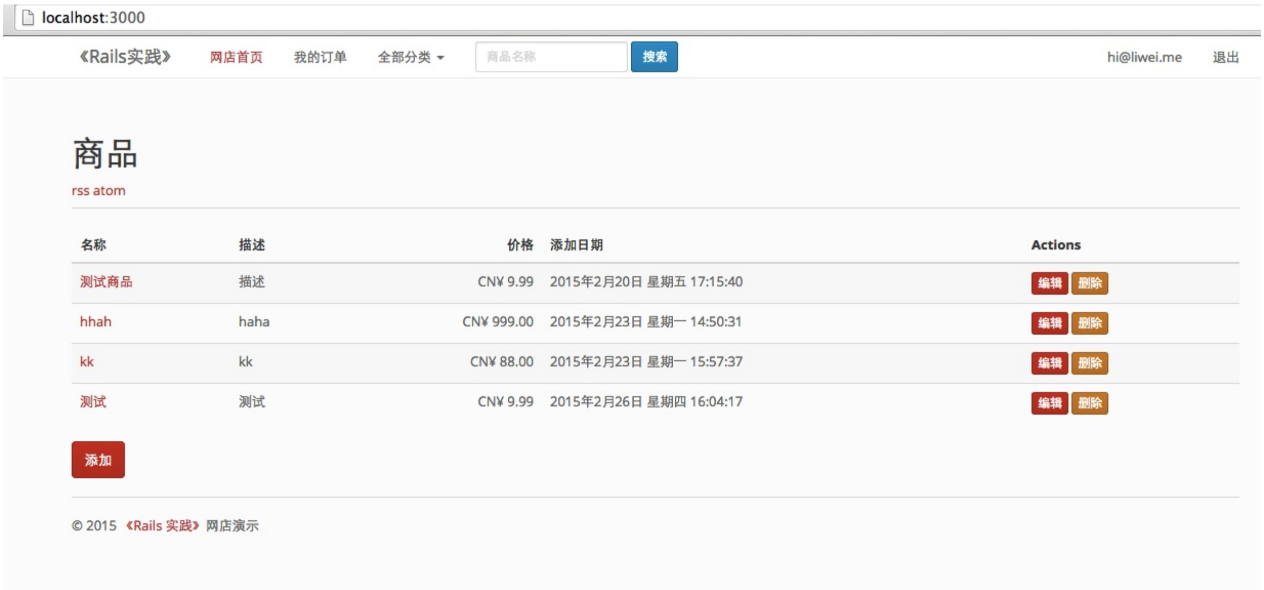
```
def create
  sleep 10
  @product = Product.new(product_params)
  ...
```

更多 ujs 支持的属性，我们在 这里 看到。

3.3.2 无刷新页面的操作

ujs 给我们带来的一些便利还不止这些，我们来点复杂的：在不刷新页面的情形下，添加一个商品，并显示在列表中。

我们现在的列表页是这样的：

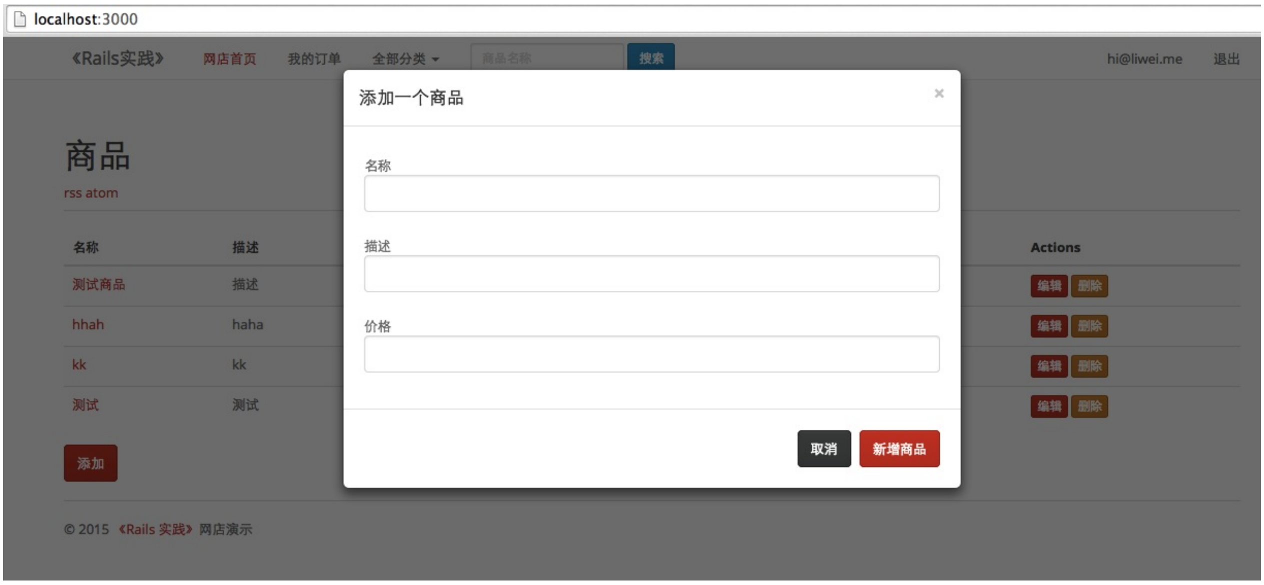


现在点击添加，我们会进入到 `http://localhost:3000/products/new` ，我们并不改变它，毕竟在某些 js 失效的情形下，点击这个按钮还是要跳转到 new 页面的。

我们希望给页面增加一个表单，来输入新商品的信息，在这之前，我们想更酷一点，我们使用 `modal` 来显示这个表单：

```
<%= link_to t('.new', :default => t("helpers.links.new")), new_product_path, :class =>
 'btn btn-primary', data: {toggle: "modal", target: "#productForm"} %>
```

ujs 允许我们在 link 上增加额外的属性，当我们再次点击 `添加` 按钮时：



当然我做了其他一些修改，你可以在 这里 找到完整的代码。

为了产生一个 ajax 的请求，我们在表单上增加一个参数 `remote: true` ：

```
<%= form_for @product, remote: true, :html => { :class => 'form-horizontal' } do |f| %
>
```

这时，ujs 将会调用 `jQuery.ajax()` 提交表单，此时的请求是一个 `text/javascript` 请求，Rails 会返回给我们相应的结果，在我们的 action 里，增加这样的声明：

```
respond_to do |format|
  if @product.save
    format.html {...}
    format.js
  else
    format.html {...}
    format.js
  end
end
```

在保存（save）成功时，我们返回给视图（view）一个 js 片段，它可以在浏览器端执行。

我们创建一个新文件 `app/views/products/create.js.erb` ，在这里，我们将新添加商品，显示在上面的列表中。

```
$('#productsTable').prepend('<%= j render(@product) %>');
$('#productFormModal').modal('hide');
```

我们使用 `.js.erb` 的文件，方便我们在 js 文件里插入 erb 的语法。

我们将一行商品信息使用 `prepend` 方法，插入到 `productsTable` 的最上面，`j` 方法将我们的字符串转换成 js 片段。

好了，你可以试一试效果了。

你可能也像我一样做了一些测试，导致插入了很多测试数据，为了继续不刷新页面就完成删除操作，我们给 `删除` 按钮上也增加一个 ajax 调用。

我们先给每一行记录，增加一个唯一的 ID 标识，通常使用“名字 + id”的形式，我们还需要给删除连接增加 `remote: true` 属性，我们编辑 `app/views/products/_product.html.erb` ：

```
<tr id="product_<%= product.id %>">
...
<%= link_to "删除", product, :method => :delete, remote: true, :data => { :confirm => "
点击确定继续" }, :class => 'btn btn-danger btn-xs' %>
```

我们再增加一个文件以返回 js 片段给浏览器执行 `app/views/products/destroy.js.erb` ：

```
$('#product_<%= @product.id %>').fadeOut();
```

你可以再试试看。

现在，我们看一下添加商品时的返回结果：

```
$('#productsTable').prepend('<tr id=\"product_14\">\n  <td><a href=\"/products/14\">kk
k<\/a><\/td>\n  <td>jjj<\/td>\n  <td class=\"text-right\">CN¥ 999.00<\/td>\n  <td>2015
年2月26日 星期四 23:57:55<\/td>\n  <td>\n    <a class=\"btn btn-primary btn-xs\" href=\"
/products/14/edit\">编辑<\/a>\n    <a data-confirm=\"点击确定继续\" class=\"btn btn-dange
r btn-xs\" data-remote=\"true\" rel=\"nofollow\" data-method=\"delete\" href=\"/produc
ts/14\">删除<\/a>\n  <\/td>\n<\/tr>\n');
$('#productFormModal').modal('hide');
```

这里面大部分代码是不必要的HTML代码，如何让我们的返回结果更简洁呢？我们现在发送个是 `text/javascript` 请求，返回给我们的是 js 片段。下一节我们发送 'json' 请求，我们在浏览器端使用 js 处理返回的 json 数据。

3.3.3 json 数据的页面处理

为了和添加商品区分开，我们在修改商品时，使用 `json` 来处理数据，而且也在一个 `modal` 中完成。

```
<%= link_to t('.edit', :default => t("helpers.links.edit")), edit_product_path(product
), remote: true, data: { type: 'json' }, :class => 'btn btn-primary btn-xs editProduct
Link' %>
```

我们给编辑链接，增加了 `remote: true, data: { type: 'json' }`，这时我们没有打开 `modal`，我们把 js 代码写在 coffeescript 中。

我们新建一个文件，`app/assets/javascripts/products.coffee`。这个文件我们只在商品页面使用，所以不必把它放到 `simplex.js` 中，现在我们只在商品的 `index.html.erb` 中使用它，所以：

```
<%= content_for :page_javascript do %>
<%= javascript_include_tag "products" %>
...
```

当我们点击编辑按钮时，我们期望几件事：

1. 打开 `modal` 层，显示编辑表单
2. 读取这个商品的信息（json 格式），把需要编辑的内容填入表单

好，我们写上这部分代码：

```
jQuery ->
  $(".editProductLink")
    .on "ajax:success", (e, data, status, xhr) ->
      $('#alert-content').hide() [1]
      $('#editProductFormModal').modal('show') [2]
      $('#editProductName').val(data['name']) [3]
      $('#editProductDescription').val(data['description']) [3]
      $('#editProductPrice').val(data['price']) [3]
      $("#editProductForm").attr('action', '/products/'+data['id']) [4]
```

- [1] 我们隐藏错误信息提示框
- [2] 显示层
- [3] 填入编辑的信息
- [4] 更改表单提交的地址

再来看看我们的编辑表单：

```
...
<%= form_tag "", method: :put, remote: true, data: { type: "json" }, id: "editProductF
orm", class: "form-horizontal" do %>
...
<%= text_field_tag "product[name]", "", :class => 'form-control', id: "editProductName
", required: true %>
...
<%= text_field_tag "product[description]", "", :class => 'form-control', id: "editProd
uctDescription" %>
...
<%= text_field_tag "product[price]", "", :class => 'form-control', id: "editProductPri
ce" %>
...
```

我们让表单提交的地址，可以根据选择的商品而改变，同时我们设定它的 type 为 json 格式。

我们为每一个输入框，设定了 ID，这样，我们用读取的 json 信息，分别填入对应的编辑框内。

然后，我们改动一下 controller 中的方法：

```
def edit
  respond_to do |format
    format.html
    format.json { render json: @product, status: :ok, location: @product } [1]
  end
end
```

- [1] 我们让 edit 方法，返回给我们商品的 json 格式信息。

```
def update
  respond_to do |format|
    if @product.update(product_params)
      format.html { redirect_to @product, notice: 'Product was successfully updated.'
}
      format.json [1]
    else
      format.html { render :edit }
      format.json { render json: @product.errors.full_messages.join(', '), status: :er
ror } [2]
    end
  end
end
```

- [1] 我们让 update 方法，可以接受 json 的请求，
- [2] 当 update 失败时，我们需要把失败的原因告诉客户端，它也是 json 格式的。

当我们需要考虑 update 方法会有成功和失败两种可能时，我们的 ajax 调用，就要这样来写了：

```
$("#editProductForm")
  .on "ajax:success", (e, data, status, xhr) ->
    $('#editProductFormModal').modal('hide') [1]
    $('#product_'+data['id']+'_name').html(  data['name'] ) [2]
    $('#product_'+data['id']+'_description').html(  data['description'] ) [2]
    $('#product_'+data['id']+'_price').html(  data['price'] ) [2]
  .on "ajax:error", (e, xhr, status, error) ->
    $('#alert-content').show() [3]
    $('#alert-content #msg').html( xhr.responseText ) [4]
```

- [1] 我们隐藏这个层
- [2] 当成功的时候，我们把修改好的信息，放回到我们的页面中
- [3] 当失败的时候，我们显示个错误信息提示框
- [4] 我们向这个框内，填入信息

更多 controller 的介绍，后面章节还会有，这里我们要了解的是，我们页面拿到的信息，不再是 js 片段，而是 json 格式的数据。

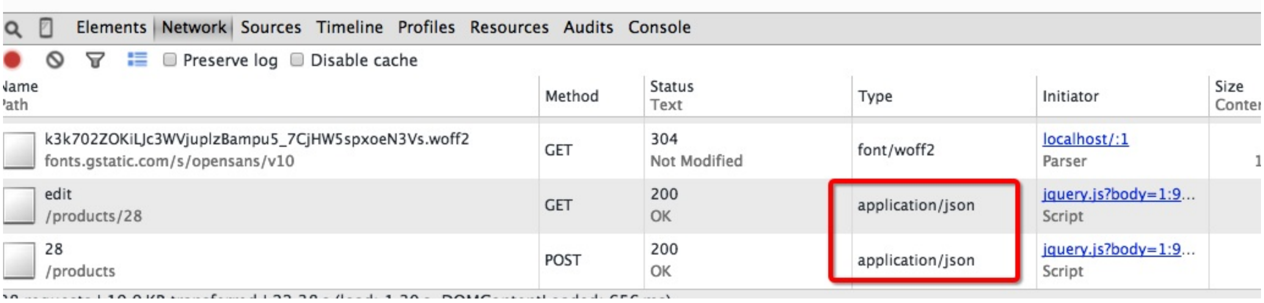
当我们处理大量数据的时候，json 明显要比 js 片段更节省传输空间，我们也可以把处理动作写到独立的 js 文件中，不过，json 格式返回给我们的，是 9.9，而我们页面显示的是格式化后的 `CN¥ 9.90` ，如果我们想把处理好格式的数据返还回来，该如何处理呢？

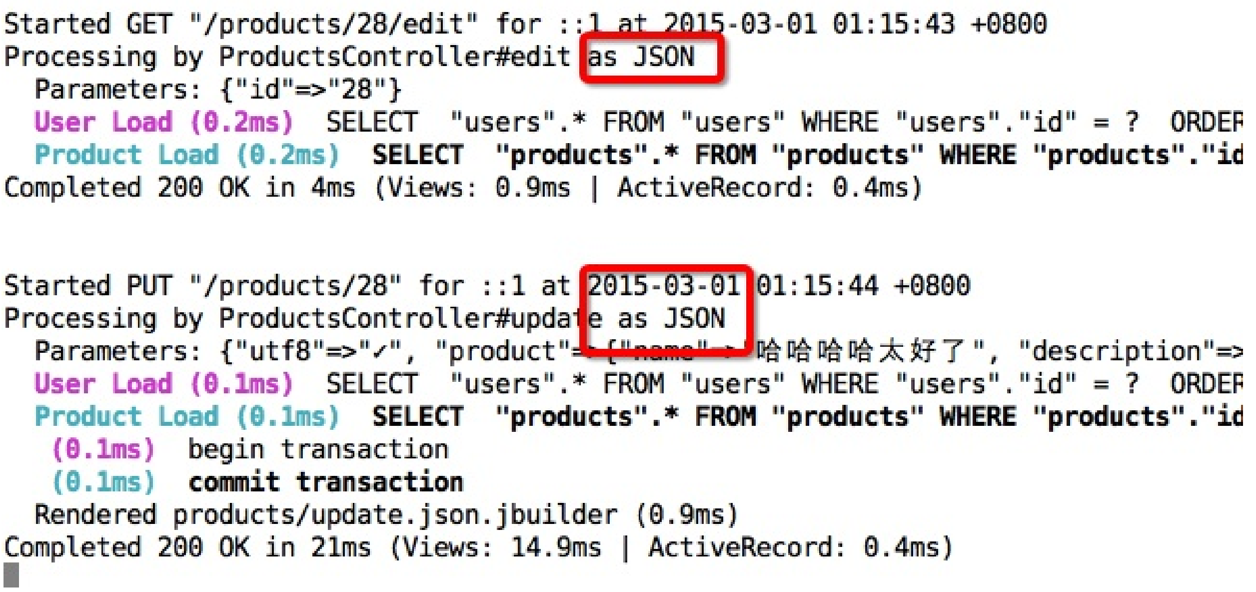
我们可以使用 jbuilder 做这件事，我们新建一个 `update.json.jbuilder` ：

```
json.id @product.id
json.name link_to @product.name, product_path(@product) [1]
json.description @product.description
json.price number_to_currency(@product.price) [2]
```

- [1] 我们把链接的地址用辅助方法生成
- [2] 我们用 number_to_currency 方法把价格格式化，这里可以使用辅助方法

如何知道我们的确使用的是 json 数据呢？我们可以查看浏览器的控制台，或者查看命令行的 log 输出。





在 这里 可以找到我调试好的代码。

在实践开发中，我们会从服务端拿到很多的内容，比如几十条订单信息，我们可以用上面的方法把它们显示到页面上，也可以使用 http://handlebarsjs.com/ 这种模板引擎，使页面和逻辑更加的独立，清晰。当我们面对少量的内容时，js 片段要比写一大堆 coffeescript 来的更省事些。所以，我们在确定选用哪种方式处理，要看我们面对的是怎样的问题。

最后附上两个附表。

附表一，当我们 `render json:..., status: :ok, ...` 时，status 和符号的对应，可以在这里找到，一般我们用 :ok, :create, :success, :error 就足够了。

Response Class	HTTP Status Code	Symbol
Informational	100	:continue
	101	:switching_protocols
	102	:processing
Success	200	:ok
	201	:created
	202	:accepted
	203	:non_authoritative_information
	204	:no_content
	205	:reset_content
	206	:partial_content
	207	:multi_status
	208	:already_reported
	226	:im_used
Redirection	300	:multiple_choices
	301	:moved_permanently
	302	:found
	303	:see_other
	304	:not_modified
	305	:use_proxy
	306	:reserved
	307	:temporary_redirect
	308	:permanent_redirect
Client Error	400	:bad_request
	401	:unauthorized
	402	:payment_required
	403	:forbidden
	404	:not_found
	405	:method_not_allowed
	406	:not_acceptable
	407	:proxy_authentication_required
	408	:request_timeout

	409	:conflict
	410	:gone
	411	:length_required
	412	:precondition_failed
	413	:request_entity_too_large
	414	:request_uri_too_long
	415	:unsupported_media_type
	416	:requested_range_not_satisfiable
	417	:expectation_failed
	422	:unprocessable_entity
	423	:locked
	424	:failed_dependency
	426	:upgrade_required
	428	:precondition_required
	429	:too_many_requests
	431	:request_header_fields_too_large
Server Error	500	:internal_server_error
	501	:not_implemented
	502	:bad_gateway
	503	:service_unavailable
	504	:gateway_timeout
	505	:http_version_not_supported
	506	:variant_also_negotiates
	507	:insufficient_storage
	508	:loop_detected
	510	:not_extended
	511	:network_authentication_required

附表二：ajax 的回调方法，我们使用了 :success 和 :error，当然还有其他的一些，我们需要了解下。

event name	extra parameters *	when
`ajax:before`		before the whole ajax business , aborts if stopped
`ajax:beforeSend`	[event, xhr, settings]	before the request is sent, aborts if stopped
`ajax:send`	[xhr]	when the request is sent
`ajax:success`	[data, status, xhr]	after completion, if the HTTP response was a success
`ajax:error`	[xhr, status, error]	after completion, if the server returned an error **
`ajax:complete`	[xhr, status]	after the request has been completed, no matter what outcome
`ajax:aborted:required`	[elements]	when there are blank required fields in a form, submits anyway if stopped
`ajax:aborted:file`	[elements]	if there are non-blank input:file fields in a form, aborts if stopped

3.4 模板引擎的使用

概要：

本课时结合商品页面，讲解如何使用简洁安全的模板引擎，以及如何更改邮件模板。

知识点：

1. haml
2. slim
3. liquid
4. 邮件模板

正文

3.4.1 haml

前面的章节里，我们一直使用 erb 作为视图模板，erb 可以让我们在 html 中签入 Ruby 代码。这样做的好处是，我们拿到的页面和设计师提供的页面几乎无任何差别，可以直接增加上我们用 Ruby 写的的逻辑。稍微不好的一点是，html 太多了，稍微处理不好，会缺失标签，而且不易察觉。

这时我们可以使用其他一些方案，haml 是比较常用的一个。

我们在 Gemfile 中安装 haml：

```
gem 'haml'
```

我们看一下用 haml 写的代码：

```
%section.container
  %h1= post.title
  %h2= post.subtitle
  .content
    = post.content
```

下面是 erb 的写法。

```
<section class="container">
  <h1><%= post.title %></h1>
  <h2><%= post.subtitle %></h2>
  <div class="content">
    <%= post.content %>
  </div>
</section>
```

可见 haml 节省了我们大量的代码，而且更接近 Ruby 语法。我们看几个 haml 常用的写法：

```
.title= "I am Title"
#title= "I am Title"
```

它会输出为：

```
 <div class="title">I am Title</div>
 <div id="title">I am Title</div>
```

下面是显示 ul 列表，注意，haml 的缩进是2个空格：

```
%ul
  %li Salt
  %li Pepper
```

这是循环的例子：

```
- (42...47).each do |i|
  %p= i
%p See, I can count!
```

我们如果想在项目里使用 haml 文件，只需要创建一个 `xxx.html.haml` 文件即可，这是一个完整的 haml 例子：

```
!!!
%html{html_attrs}
  %head
    %title Hampton Catlin Is Totally Awesome
    %meta{"http-equiv" => "Content-Type", :content => "text/html; charset=utf-8"}
  %body
    %h1
      This is very much like the standard template,
      except that it has some ActionView-specific stuff.
      It's only used for benchmarking.
    .crazy_partials= render :partial => 'templates/av_partial_1'
    / You're In my house now!
    .header
      Yes, ladies and gentileman. He is just that egotistical.
      Fantastic! This should be multi-line output
      The question is if this would translate! Ahah!
      = 1 + 9 + 8 + 2 #numbers should work and this should be ignored
    #body= " Quotes should be loved! Just like people!"
    - 120.times do |number|
      - number
    Wow.|
    %p
      = "Holy cow        " + |
        "multiline       " + |
        "tags!           " + |
        "A pipe (|) even!"   |
      = [1, 2, 3].collect { |n| "PipesIgnored|" }
      = [1, 2, 3].collect { |n|     |
          n.to_s                    |
        }.join("|")                 |
    %div.silent
      - foo = String.new
      - foo << "this"
      - foo << " shouldn't"
      - foo << " evaluate"
      = foo + " but now it should!"
      -# Woah crap a comment!

    -# That was a line that shouldn't close everything.
    %ul.really.cool
      - ('a'..'f').each do |a|
        %li= a
    #combo.of_divs_with_underscore= @should_eval = "with this text"
    = [ 104, 101, 108, 108, 111 ].map do |byte|
      - byte.chr
    .footer
      %strong.shout= "This is a really long ruby quote. It should be loved and wrapped
 because its more than 50 characters. This value may change in the future and this tes
t may look stupid. \nSo, I'm just making it *really* long. God, I hope this works"
```

这个文件来自 这里，你可以在这里找到它的源码。

在 这里 还有一份文档。

如果打算把现有的 erb 转成 haml，可以使用 haml 提供的一个命令行工具 html2haml，我们在 Gemfile 里安装台：

```
gem 'html2haml'
```

在命令行里，直接使用它：

```
% html2haml -e index.erb index.haml
```

-e 参数，可以把 erb 模板转成 haml。

这里有一个 网站，是在线把 html/erb 转成 haml。如果需要把 haml 转回 erb 模板，可是试试 这个网站。

为了方便的使用 haml，尤其在使用 scaffold 或者 generate 创建文件的时候，自动创建 haml，而不是 erb，可以安装 haml-rails：

```
gem "haml-rails"
```

它为我们提供了一个快速转换的工具：

```
rake haml:erb2haml
```

它可以把所有 views 文件夹下的 erb 模板，转成 haml。

3.4.2 slim

和 haml 类似，slim 更加的简洁，从它的官网首页可以看到它的代码风格，而且比 haml 又少了一些分隔符。

```
doctype html
html
  head
    title Slim Examples
    meta name="keywords" content="template language"
    meta name="author" content=author
    javascript:
      alert('Slim supports embedded javascript!')

  body
    h1 Markup examples

    #content
      p This example shows you how a basic Slim file looks like.

      == yield

      - unless items.empty?
        table
          - for item in items do
            tr
              td.name = item.name
              td.price = item.price
      - else
        p
         | No items found.  Please add some inventory.
           Thank you!

    div id="footer"
      = render 'footer'
      | Copyright © #{year} #{author}
```

这是官网首页给出的代码示例，这里有它详尽的使用手册。

slim 为我们提供了两个工具：html2slim 可以把 html/erb 转换成 slim，haml2slim 把 haml 转成 slim。

和 haml-rails 一样，slim-rails 可以默认生成 slim 模板。

在这里，有一个 在线工具，把 html 转换成 slim。

3.4.3 liquid

上面两个模板引擎（template engine）是针对开发者的，因为我们编写的代码是不会交付给使用者的，但是，如果我们需要把页面开放给使用者随意编辑，以上提到的 erb，haml，slim 是绝对不可以的，因为使用者可以在页面里这么写：

```
<%= User.destroy_all %>
```

那么，如何给使用者一个安全的模板来自由编辑呢？liquid 是一个很好的方案。liquid 是著名的电商网站 Shopify 设计并开源的安全模板引擎。

liquid 不允许执行危险的代码，所以可以随意交给使用者编辑并且直接渲染成页面，它还可以保存到数据里，这样可以实现在线编辑模板，它将逻辑代码和表现代码分开，如果你熟悉 php 的 smarty 模板，那么你会发现 liquid 就是 Ruby 版的 smarty。

我们看一个例子：

```
<ul id="products">
  {% for product in products %}
    <li>
      <h2>{{ product.name }}</h2>
      Only {{ product.price | price }}
      {{ product.description | prettyprint | paragraph }}
    </li>
  {% endfor %}
</ul>
```

这是我们的 liquid 模板，我们把 Product 的 Model 也改写一下，让 liquid 可以读取它的属性：

```
class Product < ActiveRecord::Base
  def to_liquid
    {
      "name" => name,  [1] 这里要用 string 的写法，不要使用 symbol。
      "price" => price
    }
  end
end
```

我们来渲染（Render）这个模板：

```
require "liquid"
template = Liquid::Template.parse(template) # template 就是上面的代码，可以直接从数据库读取出来。
template.render('products' => Product.all) # 传入 products 变量，这是由我们控制传入的，用户可以在模板中随意调用。
```

liquid 的源码在 https://github.com/Shopify/liquid，在 https://github.com/Shopify/liquid/wiki/Liquid-for-Designers，有非常详尽的使用方法。

和 haml-rails，slim-rails 一样，liquid 也有自己的：

```
gem 'liquid-rails'
```

它可以方便的使用 Rails 中的 Helper 和 Tag，实现 Drop Class，并且编写 Rspec 测试。

Rails 使用 Tilt 这个 Gem 来处理各种模板引擎，Tilt 是一个接口，支持几十个模板引擎，我们只是提到了其中较常用的三个。

使用 tilt 方便我们在 Rails 集成各种模板引擎，不过实际开发的时候，我们要注意他们的效率问题。这里有一份测试数据：

```
# Linux + Ruby 1.9.2, 1000 iterations

                      user     system      total        real
(1) erb             0.680000   0.000000   0.680000 (  0.810375)
(1) erubis          0.510000   0.000000   0.510000 (  0.547548)
(1) fast erubis     0.530000   0.000000   0.530000 (  0.583134)
(1) slim            4.330000   0.020000   4.350000 (  4.495633)
(1) haml            4.680000   0.020000   4.700000 (  4.747019)
(1) haml ugly       4.530000   0.020000   4.550000 (  4.592425)

(2) erb             0.240000   0.000000   0.240000 (  0.235896)
(2) erubis          0.180000   0.000000   0.180000 (  0.185349)
(2) fast erubis     0.150000   0.000000   0.150000 (  0.154970)
(2) slim            0.050000   0.000000   0.050000 (  0.046685)
(2) haml            0.490000   0.000000   0.490000 (  0.497864)
(2) haml ugly       0.420000   0.000000   0.420000 (  0.428596)

(3) erb             0.030000   0.000000   0.030000 (  0.033979)
(3) erubis          0.030000   0.000000   0.030000 (  0.030705)
(3) fast erubis     0.040000   0.000000   0.040000 (  0.035229)
(3) slim            0.040000   0.000000   0.040000 (  0.036249)
(3) haml            0.160000   0.000000   0.160000 (  0.165024)
(3) haml ugly       0.150000   0.000000   0.150000 (  0.146130)

(4) erb             0.060000   0.000000   0.060000 (  0.059847)
(4) erubis          0.040000   0.000000   0.040000 (  0.040770)
(4) slim            0.040000   0.000000   0.040000 (  0.047389)
(4) haml            0.190000   0.000000   0.190000 (  0.188837)
(4) haml ugly       0.170000   0.000000   0.170000 (  0.175378)

1. Uncached benchmark. Template is parsed every time.
   Activate this benchmark with slow=1.

2. Cached benchmark. Template is parsed before the benchmark.
   The ruby code generated by the template engine might be evaluated every time.
   This benchmark uses the standard API of the template engine.

3. Compiled benchmark. Template is parsed before the benchmark and
   generated ruby code is compiled into a method.
   This is the fastest evaluation strategy because it benchmarks
   pure execution speed of the generated ruby code.

4. Compiled Tilt benchmark. Template is compiled with Tilt, which gives a more
   accurate result of the performance in production mode in frameworks like
   Sinatra, Ramaze and Camping. (Rails still uses its own template
   compilation.)
```

该数据来自：https://ruby-china.org/topics/634

选择哪个模板引擎，还是直接使用 erb，需要视情况而定了。

3.4.4 devise 的邮件模板

Devise 除了提供用户注册和登录功能，还可以通过邮件激活用户。这里，我们可以自己定义邮件模板中的内容。

我们修改一下 User 中关于 Devise 的配置：

```
devise :database_authenticatable, :registerable,
       :recoverable, :rememberable, :trackable, :validatable,
       :confirmable
```

我们增加了一个新的选项：:confirmable。

我们配置下邮件发送的信息，这里我们只在开发环境（development）配置，我们打开 `config/environments/development.rb`：

```
  config.action_mailer.default_url_options = { host: 'localhost', port: 3000 }
  config.action_mailer.delivery_method = :smtp
  config.action_mailer.smtp_settings = {
    address:              'smtp.163.com',
    port:                 25,
    domain:               '...',
    user_name:            '...@163.com',
    password:             '...',
    authentication:       :plain,
    enable_starttls_auto: true
  }
```

同时，我们还需要修改 user 创建时的 migration 文件，打开 `db/migrate/xxxx_devise_create_users.rb`，我们取消注释这个部分：

```
## Confirmable
  t.string   :confirmation_token
  t.datetime :confirmed_at
  t.datetime :confirmation_sent_at
  t.string   :unconfirmed_email # Only if using reconfirmable
```

注意，xxxx 是日期时间戳，表示这个文件创建的日期。devise 已经为我们添加了 `confirmable` 需要的字段，我们不必自己添加。这里有一个问题，我们已经运行过数据库文件了，这里是修改旧文件，我们不能直接更新文件，这里我们可以删掉旧的数据库，其实在开发环境，我们可以经常重建数据库：

```
rake db:drop
rake db:create
rake db:migrate
rake db:seed
```

还记得 `db/seeds.rb` 这个文件吧，我们可以把一些默认数据写到 seed 里，或者一些测试数据，比如我们添加50个商品信息，来测试分页等效果是否正确，或者初始化几十个商品类别等。这样，重建数据库时，不必担心默认数据的丢失。

我们再编辑下 devise 的配置文件，它在 `config/initializers/devise.rb` ：

```
config.mailer_sender = '...@163.com'
...
config.mailer = 'Devise::Mailer'
```

我们重新启动 Rails 服务，注册一个账号，这时我们观察终端，可以看到邮件发送的信息：

```
Sent mail to hi@liwei.me (2468.6ms)
Date: Mon, 02 Mar 2015 14:04:57 +0800
From: master@...
Reply-To: master@...
To: hi@liwei.me
Message-ID: <54f3fd89f0f84_62213ffdf44a34e08208d@macbook.local.mail>
Subject: =?UTF-8?Q?=E6=9D=A5=E8=87=AAezcms=E7=9A=84=E6=B3=A8=E5=86=8C=E7=A1=AE=E8=AE=A
4=E9=82=AE=E4=BB=B6?=
Mime-Version: 1.0
Content-Type: text/html;
 charset=UTF-8
Content-Transfer-Encoding: 7bit

 <p>Welcome hi@liwei.me!</p>

 <p>You can confirm your account email through the link below:</p>

 <p><a href="http://localhost:3000/users/confirmation?confirmation_token=FNMGvy_VnNfyh
HKz_LKY">Confirm my account</a></p>
```

我们页面上，也会得到这样的提示：



为了让我们的邮件看起来更友好，我们编辑 `app/views/users/mailer/confirmation_instructions.html.erb` ：

```
 <p>你好 <%= @email %>!</p>

 <p>请点击下面的确认链接，验证您的邮箱:</p>

 <p><%= link_to "验证我的邮箱", confirmation_url(@resource, confirmation_token: @token)
%></p>
```

你可以再试试看。不过这种配置可能会被当做垃圾邮件拒收，或者直接被放到垃圾邮件中。在后面的章节里，我们会介绍其他的方式发送邮件。

如果你不能通过邮件激活这个用户，比如那些在 seed 中添加的用户，没关系， `rails c` 进入控制台：

```
u = User.last [1]
u.confirm! [2]
```

- [1] 找到这个用户，更多方法在下一章陆续介绍
- [2] `confirm!` 方法激活用户

更多邮件配置，可以查看 http://guides.rubyonrails.org/configuring.html#configuring-action-mailer

第四章 Rails 中的模型

课程概要：

本课程讲解Rails 模型（Model）中基本的 CRUD 操作、模型间的关联关系、属性校验、回调以及编写 Rspec 测试的方法，并完成网店的数据库模型设计。

知识点：

1. CRUD
2. 数据库迁移（Migration）
3. 表间关联（Relations）
4. 属性校验（Validates）
5. 回调（Callback）

课程背景

模型（Model）是 MVC 架构中的 M，代表数据库，通过对模型的学习，可以了解 Rails 是如何实现数据库操作的。

4.1 模型的基础操作

概要：

本课时讲解模型的基础操作，数据迁移，常用的 CRUD 方法，在数据查询时，如何避免 N+1 问题，如何使用 scope 包装查询条件，编写模型 Rspec 测试。

知识点：

1. Active Record
2. Migration
3. CRUD

正文

4.1.1 Active Record 简介

Active Record 模式，是由 Martin Fowler 的《企业应用架构模式》一书中提出的，在该模式中，一个 Active Record（简称 AR）对象包含了持久数据（保存在数据库中的数据）和数据操作（对数据库里的数据进行操作）。

对象关系映射（Object-Relational Mapping，简称 ORM），是将程序中的对象（Object）和关系型数据库(Relational Database)的表之间进行关联。使用 ORM 可以方便的将对象的 `属性` 和 `关联关系` 保存入数据库，这样可以不必编写复杂的 SQL 语句，而且不必担心使用的是哪种数据库，一次编写的代码可以应用在 Sqlite，Mysql，PostgreSQL 等各种数据库上。

Active Record 就是个 ORM 框架。

所以，我们可以用 Actice Record 来做这几件事：

- 表示模型（Model）和模型数据
- 表示模型间的关系（比如一对多，多对多关系）
- 通过模型间关联表示继承层次
- 在保存如数据库前，校验模型（比如属性校验）
- 用 `面向对象` 的方式处理数据库

4.1.2 Active Record 中的约定

Rails 中使用了 ActiveRecord 这个 Gem，使用它可以不必去做任何配置（大多数情况是这样的），还记得 Rails 的两个哲学理念之一么：`约定优于配置` 。（另一个是 `不要重复自己` ，这是 Dave Thomas 在《程序员修炼之道》一书里提出的。）

那么，我们讲两个 Active Record 中的约定：

4.1.2.1 命名约定

- 数据表名：复数，下划线分隔单词（例如 book_clubs）
- 模型类名：单数，每个单词的首字母大写（例如 BookClub）

比如：

模型（**Class**）	数据表（**Schema**）
Post	posts
LineItem	line_items
Deer	deers
Mouse	mice
Person	people

单词在单复数转换时，是按照英文语法约定的。

4.1.2.2 Schema 约定

注：数据库中的 Schema，指数据库对象集合，可以被用户直接使用。Schema 包含数据的逻辑结构，用户可以通过命名调用数据库对象，并且安全的管理数据库。

- 外键 - 使用 singularized_table_name_id 形式命名，例如 item_id，order_id。创建模型关联后，Active Record 会查找这个字段；
- 主键 - 默认情况下，Active Record 使用整数字段 id 作为表的主键。使用 Active Record 迁移创建数据表时，会自动创建这个字段；

在数据库字段命名的时候，有几个特殊意义的名字，尽量回避：

- created_at - 创建记录时，自动设为当前的时间戳
- updated_at - 更新记录时，自动设为当前的时间戳
- lock_version - 在模型中添加乐观锁定功能
- type - 让模型使用单表继承，给字段命名的时候，尽量避开这个词
- (association_name)_type - 多态关联的类型
- (table_name)_count - 保存关联对象的数量。例如，posts 表中的 comments_count 字段，Rails 会自动更新该文章的评论数

4.1.3 数据库迁移（Migration）

在我们使用 scaffold 创建资源的时候，或者使用 generate 创建 model 的时候，Rails 会给我们自动创建一个数据库迁移文件，它在 `db/migrate` 中，它的前缀是时间戳，他们按照时间的先后顺序排列，当运行数据库迁移时，他们按照时间顺序先后被执行。

新创建的迁移文件，我们使用 `rake db:migrate` 命令执行它（们），这里会判断，哪个迁移文件是还没有被执行的。

如果我们对执行过的迁移操作不满意，我们可以回滚这个迁移：

```
rake db:rollback [1]
rake db:rollback STEP=3 [2]
```

[1] 回滚最近的一个迁移

[2] 回滚指定的迁移个数

回滚之后，迁移停留在回滚到的那个位置的，schema 也会更新到那个位置时的状态。比如，我们上一次迁移执行了5个文件，我们回滚的时候，是一个个文件回滚的，所以我们指定 STEP=5，才能把刚才迁移的5个文件回滚。

在我们开发代码的过程中，有是会因为失误少写了一个字段，我们回滚之后，在迁移文件中把它加上，然后，我们 `rake db:migrate` 再次运行。不过，`rake db:migrate:redo [STEP=3]` 直接回滚然后再次运行迁移，这样会方便些。

这种回滚操作适合开发过程中，出现了新的想法，而回滚最近连续的几个迁移。

如果我们想回滚很久以前的某个操作，而且在那个迁移之后，我们已经执行了多个迁移。这时该如何处理呢？

如果在开发阶段，我们干脆 `rake db:drop`，`rake db:create`，`rake db:migrate`。但是在生产环境，我们决不能这么做，这时我们要针对需求，编写一个迁移文件：

```
class ChangeProductsPrice < ActiveRecord::Migration
  def change
    reversible do |dir|
      change_table :products do |t|
        dir.up   { t.change :price, :string }
        dir.down { t.change :price, :integer }
      end
    end
  end
end
```

或者：

```
class ChangeProductsPrice < ActiveRecord::Migration
  def up
    change_table :products do |t|
      t.change :price, :string
    end
  end

  def down
    change_table :products do |t|
      t.change :price, :integer
    end
  end
end
```

`up` 是向前迁移到最新的，`down` 用于回滚。

我们创建一个 model 的时候，会自动创建它的 migration 文件，我们还可以使用 `rails g migration XXX` 的方法，添加自定义的迁移文件。如果我们的命名是 "AddXXXToYYY" 或者 "RemoveXXXFromYYY" 时，会自动为我们添加字符类型的字段，比如我为 variant 添加一个 color 字段：

```
rails g migration AddColorToVariants color:string
```

它的内容是：

```
class AddColorToVariants < ActiveRecord::Migration
  def change
    add_column :variants, :color, :string
  end
end
```

4.1.4 CRUD

CRUD并不是一个 Rails 的概念，它表示系统（业务层）和数据库（持久层）之间的基本操作，简单的讲叫“增（C）删（D）改（U）查（R）”。

我们已经使用 scaffold 命令创建了资源：商品（product），我们现在使用 `app/models/product.rb` 来演示这些操作。

首先，我们需要让 Product 类继承 ActiveRecord：

```
class Product < ActiveRecord::Base
end
```

这样，Product 类就可以操作数据库了，是不是很简单。

4.1.5 创建记录

我们使用 Product 类，向数据添加一条记录，我们先进入 Rails 控制台：

```
% rails c
Loading development environment (Rails 4.2.0)
> Product.create [1]
   (0.2ms)  begin transaction [2]
  SQL (2.8ms)  INSERT INTO "products" ("created_at", "updated_at") VALUES (?, ?)  [["c
reated_at", "2015-03-14 16:23:44.640578"], ["updated_at", "2015-03-14 16:23:44.640578"
]]
   (0.8ms)  commit transaction [2]
 => #<Product id: 1, name: nil, description: nil, price: nil, created_at: "2015-03-14
16:23:44", updated_at: "2015-03-14 16:23:44"> [3]
```

这里，我贴出了完整的代码。

[1]，我们使用了 Product 的类方法 create，创建了一条记录。我们还有其他的方法保存记录。

[2]，begin 和 commit ，将我们的数据保存入数据库。如果在保存的时候出现错误，比如属性校验失败，抛出异常等，不会将记录保存到数据库。

[3]，我们拿到了一个 Product 类的实例。

除了类方法，我们还可以使用实例的 `save` 方法，来保存记录到数据，比如：

```
> product = Product.new [1]
 => #<Product id: nil, name: nil, description: nil, price: nil, created_at: nil, updat
ed_at: nil>  [2]
> product.save [3]
   (0.1ms)  begin transaction [4]
  SQL (0.9ms)  INSERT INTO "products" ("created_at", "updated_at") VALUES (?, ?)  [["c
reated_at", "2015-03-14 16:47:26.817663"], ["updated_at", "2015-03-14 16:47:26.817663"
]]
   (9.3ms)  commit transaction [4]
 => true [5]
```

[1]，我们使用类方法 new，来创建一个实例，注意，[2] 告诉我们，这是一个没有保存到数据库的实例，因为它的 id 还是 nil。

[3] 我们使用实例方法 save，把这个实例，保存到数据库。

[4] 调用 save 后，会返回执行结果，true 或者 false。这种判断很有用，而且也很常见，如果你现在打开 `app/controllers/products_controller.rb` 的话，可以看到这样的判断：

```
if @product.save
    ...
else
    ...
end
```

那么，你可能会有个疑问，使用类方法 create 保存的时候，如果失败，会返回我们什么呢？是一个实例，还是 false？

我们使用下一章里要介绍的属性校验，来让保存失败，比如，我们让商品的名称必须填写：

```
class Product < ActiveRecord::Base
  validates :name, presence: true [1]
end
```

[1] validates 是校验命令，要求 name 属性必须填写。

好了，我们来测试下类方法 create 会返回给我们什么：

```
> product = Product.create
   (0.3ms)  begin transaction
   (0.1ms)  rollback transaction
 => #<Product id: nil, name: nil, description: nil, price: nil, created_at: nil, updat
ed_at: nil>
2.2.0 :003 >
```

答案揭晓，它返回给我们一个未保存的实例，它有一个实用的方法，可以查看哪里出了错误：

```
> product.errors.full_messages
 => ["名称不能为空字符"]
```

当然，判断一个实例是否保存成功，不必去检查它的 errors 是否为空，有两个方法会根据 errors 是否添加，而返回实例的状态：

```
person = Person.new
person.invalid?
person.valid?
```

要留意的是，invalid? 和 valid? 都会调用实例的校验。

我使用类方法和实例方法的称呼，希望没有给你造成理解的障碍，如果有些难理解，建议你先看一看 Ruby 中关于类和实例的介绍。

4.1.6 查询记录

4.1.6.1 Find 查询

数据查询，是 Rails 项目经常要做的操作，如何拿到准确的数据，优化查询，是我们要重点关注的。

查询时，会得到两种结果，一个实例，或者实例的集合（Array）。如果找不到结果，也会给有两种情况，返回 nil或空数组，或者抛出 ActiveRecord::RecordNotFound 异常。

Rails 给我们提供了这些常用的查询方法：

方法名称	含义	参数	例子	找不到时
find	获取指定主键对应的对象	主键值	Product.find(10)	异常
take	获取一个记录，不考虑任何顺序	无	Product.take	nil
first	获取按主键排序得到的第一个记录	无	Product.first	nil
last	获取按主键排序得到的最后一个记录	无	Product.last	nil
find_by	获取满足条件的第一个记录	hash	Product.find_by(name: "T恤")	nil

表中的四个方法不会抛出异常，如果需要抛出异常，可以在他们名字后面加上 `!`，比如 Product.take!。

如果将上面几个方法的参数改动，我们就会得到集合：

方法名称	含义	参数	例子	找不到时
find	获取指定主键对应的对象	主键值集合	Product.find([1,2,3])	异常
take	获取一个记录，不考虑任何顺序	个数	Product.take(2)	[]
first	获取按主键排序得到的第N个记录	个数	Product.first(3)	[]
last	获取按主键排序得到的最后N个记录	个数	Product.last(4)	[]
all	获取按主键排序得到的全部记录	无	Product.all	[]

Rails 还提供了一个 find_by 的查询方法，它可以接收多个查询参数，返回符合条件的第一个记录。比如：

```
Product.find_by(name: 'T-Shirt', price: 59.99)
```

`find_by` 有一个常用的变形，比如：

```
Product.find_by_name("Hat")
Product.find_by_name_and_price("Hat", 9.99)
```

如果需要查询不到结果抛出异常，可以使用 `find_by!` 。通常，以 `!` 结尾的方法都会抛出异常，这也是一种约定。不过，直接使用 find，会查询主索引，查询不到直接抛出异常，所以是没有 `find!` 方法的。

使用 find_by 的时候，还可以使用 sql 语句，比如：

```
Product.find_by("name = ?", "T")
```

这是一个有用的查询，当我们搜索多个条件，并且是 OR 关系时，可以这样做：

```
User.find_by("id = ? OR login = ?", params[:id], params[:id])
```

这句话还可以改写成：

```
User.find_by("id = :id OR login = :name", id: params[:id], name: params[:id])
```

或者更简洁的：

```
User.find_by("id = :q OR login = :q", q: params[:id])
```

4.1.6.2 Where 查询

集合的查找，最常用的方法是 `where` ，它可以通过多种形式查找记录：

查询形式	实例
数组（Array）查询	Product.where("name = ? and price = ?", "T恤", 9.99)
哈希（hash）查询	Product.where(name: "T恤", price: 9.99)
Not查询	Product.where.not(price: 9.99)
空	Product.none

使用 where 查询，常见的还有模糊查询：

```
Product.where("name like ?", "%a%")
```

查询某个区间：

```
Product.where(price: 5..6)
```

以及上面提到的，sql 的查询：

```
Product.where("color = ? OR price > ?", "red", 9)
```

Active Record 有多种查询方法，以至于 Rails 手册中单独列出一章来讲解，而且讲解的很细致，如果你想灵活的掌握这些数据查询方法，建议你经常阅读 Active Record Query Interface 一章，这是 中文版。

4.1.7 更新记录（Update）

和创建记录一样，更新记录也可以使用类方法和实力方法。

类方法是 update，比如：

```
Product.update(1, name: "T-Shirt", price: 23)
```

1 是更新目标的 ID，如果该记录不存在，update 会抛出 `ActiveRecord::RecordNotFound` 异常。

`update` 也可以更新多条记录，比如：

```
Product.update([1, 2], [{ name: "Glove", price: 19 }, { name: "Scarf" }])
```

我们看看它的源代码：

```
# File activerecord/lib/active_record/relation.rb, line 363
def update(id, attributes)
  if id.is_a?(Array)
    id.map.with_index { |one_id, idx| update(one_id, attributes[idx]) }
  else
    object = find(id)
    object.update(attributes)
    object
  end
end
```

如果要更新全部记录，可以使用 update_all ：

```
Product.update_all(price: 20)
```

在使用 update 更新记录的时候，会调用 Model 的 validates（校验） 和 callbacks（回调），保证我们写入正确的数据，这个是定义在 Model 中的方法。但是，update_all 会略过校验和回调，直接将数据写入到数据库中。

和 update_all 类似，update_column/update_columns 也是将数据直接写入到数据库，它是一个实例方法：

```
product = Product.first
product.update_column(:name, "")
product.update_columns(name: "", price: 0)
```

虽然为 product 增加了 name 非空的校验，但是 update_column(s) 还是可以讲数据写入数据库。

当我们创建迁移文件的时候，Rails 默认会添加两个时间戳字段，created_at 和 updated_at。

当我们使用 update 更新记录时，触发 Model 的校验和回调时，也会自动更新 updated_at 字段。但是 Model.update_all 和 model.update_column(s) 在跳过回调和校验的同时，也不会更新 updated_at 字段。

我们也可以用 save 方法，将新的属性保存到数据库，这也会触发调用和回调，以及更新时间戳：

```
product = Product.first
product.name = "Shoes"
product.save
```

4.1.8 删除记录（Destroy）

在我们接触计算机英语里，表示删除的英文有很多，这里我们用到的是 destroy, delete。

4.1.8.1 Delete 删除

使用 delete 删除时，会跳过回调，以及关联关系中定义的 `:dependent` 选项，直接从数据库中删除，它是一个类方法，比如：

```
Product.delete(1)
Product.delete([2,3,4])
```

当传入的 id 不存在的时候，它不会抛出任何异常，看下它的源码：

```
# File activerecord/lib/active_record/relation.rb, line 502
def delete(id_or_array)
  where(primary_key => id_or_array).delete_all
end
```

它使用不抛出异常的 where 方法查找记录，然后调用 delete_all。

delete 也可以是实例方法，比如：

```
product = Product.first
product.delete
```

在有具体实例的时候，可以这样使用，否则会产生 `NoMethodError: undefined method `delete' for nil:NilClass`，这在我们设计逻辑的时候要注意。

delete_all 方法和 delete 是一样的，直接发送数据删除的命令，看一下 api 文档中的例子：

```
Post.delete_all("person_id = 5 AND (category = 'Something' OR category = 'Else')")
Post.delete_all(["person_id = ? AND (category = ? OR category = ?)", 5, 'Something', '
Else'])
Post.where(person_id: 5).where(category: ['Something', 'Else']).delete_all
```

4.1.8.2 Destroy 删除

destroy 方法，会触发 model 中定义的回调（before_remove, after_remove , before_destroy 和 after_destroy），保证我们正确的操作。它也可以是类方法和实例方法，用法和前面的一样。

需要说明，delete/delete_all 和 destroy/destroy_all 都可以作用在关系查询结果，也就是（ActiveRecord::Relation）上，删掉查找到的记录。

如果你不想真正从数据库中抹掉数据，而是给它一个删除标注，可以使用 https://github.com/radar/paranoia 这个 gem，他会给记录一个 deleted_at 时间戳，并且使用 `restore` 方法把它从数据库中恢复过来，或者使用 `really_destroy!` 将它真正的删除掉。

4.2 深入模型查询

概要：

本课时讲解模型在数据查询时，如何避免 N+1问题，使用 scope 包装查询条件，编写模型 Rspec 测试。

知识点：

1. N+1
2. Scope
3. 实用的查询
4. Rspec 测试

正文

4.2.1 两个 Gem

ActiveRecord 这个 gem 中，包含了两个重要的 gem，打开它的 源代码，可以看到这两个 gem：activemodel 和 arel。

`activemodel` 为一个类增加了许多特性，比如属性校验，回调等，这在后面章节会介绍。

`arel` 是 Ruby 编写的 sql 工具，使用它，可以通过简单的 Ruby 语法，编写复杂 sql 查询，我们上面使用的例子，语法就来自 arel。arel 还可以面向多种关系型数据库。

ActiveRecord 在使用 arel 的时候，提供了一个方法：sanitize_sql。

在我们以上的讲解中，会经常传递这样的参数 `["name = ? and price=?", "foobar", 4]`，它会由 `sanitize_sql` 方法进行处理，这是一个 protected 方法，我们使用 send 来调用它：

```
Product.send(:sanitize_sql, ["name = ? and price=?", "Shoes", 4])
=> "name = 'Shoes' and price=4"
```

这是一种安全的手段，保护我们的 sql 不会被插入恶意代码。我们不必去直接使用这个方法，除非特殊情况，我们只需要按照它的格式要求来书写就可以了。

4.2.2 N+1

N+1 是查询中经常遇到的一个问题。在下一节里，我们经常使用关联关系的查询，比如，列出十个用户的同时，显示它地址中的电话：

```
users = User.limit(10)

users.each do |user|
  puts user.address.phone
end
```

这样就会造成，在 each 中又去查询数据，得到电话。这种情况会经常出现在我的列表中，所以在列表中会经常遇到 N+1 的问题。

为了避免这个问题，Rails 提供了预加载的功能，在查询的时候，使用 `includes` 来解决。上面的例子修改一下：

```
users = User.includes(:address).limit(10)

users.each do |user|
  puts user.address.phone
end
```

我们查看一下终端的输出：

```
SELECT * FROM users LIMIT 10
SELECT addresses.* FROM addresses
  WHERE (addresses.user_id IN (1,2,3,4,5,6,7,8,9,10))
```

这里只有两个 sql 查询，提高了查询效率。

4.2.3 查询中使用 Scope

当我们使用 where 查询的时候，会遇到多个条件组合查询。通常我们可以把它们都写到一个 where 的条件里，比如：

```
Product.where(name: "T-Shirt", hot: true, top: true)
```

我增加了两个条件，`hot: true` 和 `top: true` ，但是，这种条件组合只能在这里使用，在其他地方，我们还要再写一遍，这不符合 Rails 的哲学：“不要重复自己”。

Rails 提供了 scope，让我们复用查询条件：

```
class Product < ActiveRecord::Base
  scope :hot, -> { where(hot: true) }
  scope :top, -> { where(top: true) }
end
```

使用的时候，我们可以将多个 scope 组合在一起：

```
Product.top.hot.where(name: "T-Shirt")
```

`default_scope` 可以为所有查询加上它定义的查询条件，比如：

```
class Product < ActiveRecord::Base
  default_scope { where("deleted_at IS NULL") }
end
```

`default_scope` 要慎用，慎用，慎用（重要的话说三遍），在我们程序变的复杂的时候，性能往往会消耗在数据库查询上，维护已有查询时，很容易忽视 default_scope 的作用。如果使用了 default_scope，而在其他地方不得不去掉它，可以使用 unscoped，然后再附上其他查询：

```
Product.unscoped.load.top.hot
```

如果一个地方使用了某个 scope，而要在另一个地方把它的条件改变，可以使用 merge：

```
class Product < ActiveRecord::Base
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end
```

看一下它的执行结果：

```
Product.active.merge(User.inactive)
# SELECT "products".* FROM "products" WHERE "products"."state" = 'inactive'
```

4.2.4 实用的查询

4.2.4.1 sql 查询集合

我们使用where查询，得到的是 ActiveRecord::Relation 实例，它的源代码在这里。阅读这里的代码，会让你学习到更多优雅的查询方法。在查询时，我们还可以使用 sql 直接查询，如果你更熟悉 sql 语法，可以这样来查询：

```
Client.find_by_sql("SELECT * FROM clients
  INNER JOIN orders ON clients.id = orders.client_id
  ORDER BY clients.created_at desc")
# =>  [
  #<Client id: 1, first_name: "Lucas" >,
  #<Client id: 2, first_name: "Jan" >,
  # ...
]
```

这个例子来自这里。

它返回的是实例的集合，这在我们 Rails 内使用很方便，但是提供 json 格式的 api时，需要转换一下，不过我们可以用 select_all 查询，得到包含 hash 的 array：

```
Client.connection.select_all("SELECT first_name, created_at FROM clients WHERE id = '1
'")
# => [
  {"first_name"=>"Rafael", "created_at"=>"2012-11-10 23:23:45.281189"},
  {"first_name"=>"Eileen", "created_at"=>"2013-12-09 11:22:35.221282"}
]
```

4.2.4.2 pluck

pluck 可以直接在 Relation 实例的基础上，使用 sql 的 select 方法，得到字段值的集合（Array），而不用把返回结果包装成 ActiveRecord 实例，再得到属性值。在查询属性集合时，`pluck` 的性能更高。

```
Client.where(active: true).pluck(:id)
 SELECT id FROM clients WHERE active = 1
 => [1, 2, 3]

Client.distinct.pluck(:role)
 SELECT DISTINCT role FROM clients
 => ['admin', 'member', 'guest']

Client.pluck(:id, :name)
 SELECT clients.id, clients.name FROM clients
 => [[1, 'David'], [2, 'Jeremy'], [3, 'Jose']]
```

ActiveRecord 有一个类似的方法，select，比较下两者的区别：

```
Product.select(:id, :name)
  Product Load (8.5ms)  SELECT "products"."id", "products"."name" FROM "products"
 => #<ActiveRecord::Relation [#<Product id: 1, name: "f">]>
Product.pluck(:id, :name)
   (0.3ms)  SELECT "products"."id", "products"."name" FROM "products"
 => [[1, "f"]]
```

前者显示返回 AR 实例，然后取其属性值，后者直接读取数据库记录，返回数组。

pluck 只能用在查询的最后，因为它直接返回了结果，而不是 ActiveRecord::Relation。

4.2.4.3 ids

ids 返回主键集合：

```
Person.ids
=> SELECT id FROM people
```

不要被 ids 字面迷惑，它返回的是主键的集合，我们可以在 model 里设定其他字段为主键。

```
class Person < ActiveRecord::Base
  self.primary_key = "person_id"
end

Person.ids
=> SELECT person_id FROM people
```

4.2.4.4 查询记录数量

这里有四个方法，方便我们判断一个模型中的记录数量。

```
Client.exists?(1)
Client.exists?(id: [1,2,3])
Client.exists?(name: ['John', 'Sergei'])
```

`exists?` 判断记录是否存在，和它类似的方法有两个：

```
Client.exists? [1]
Client.any? [2]
Client.many? [3]
```

[1] 是否有记录 [2] 是否至少有一条记录 [3] 是否有多于一条的记录

any? 和 many? 与 exists? 不同的是，他们可以使用在 Relation 实例上，比如：

```
Article.where(published: true).any?
Article.where(published: true).many?
```

还可以接收 block：

```
person.pets.any? do |pet|
  pet.group == 'cats'
end
=> false

person.pets.many? do |pet|
  pet.group == 'dogs'
end
=> true
```

4.2.4.5 查询记录数量

下面五个方法，完全可以按照字面意义理解，并且适用于 Relation 上：

```
Client.count
Client.average("orders_count")
Client.minimum("age")
Client.maximum("age")
Client.sum("orders_count")
```

以上的例子来自 这里，闲暇的时候应该多读读这个文档，翻看源码。

4.2.5 Rspec 测试

在深入 Rails 项目开发之后，测试环节是一个重要的环节。Ruby 为我们提供了非常方便的测试框架，Rails 也可以方便的执行这些测试框架。

在 Rails 3.x 及之前的版本里，默认使用 TestUnit 框架，4.x 之后改为 MiniTest 框架。我们可以查看 test_case.rb 文件，看到其中的变化。

除了这两个测试框架，Rspec 也是经常用到的 Ruby 测试框架。

我们在 Rails 里安装 rpesc，和其他的几个 gem：

```
group :development, :test do
  gem 'rspec-rails'
  gem "factory_girl_rails"
  gem "database_cleaner"
end
```

rspec-rails 是 rspec 的 Rails 集成，在 Rails 中初始化 rspec 的命令是：

```
rails generate rspec:install
```

它会创建两个文件，和 spec 文件件。运行 rpsec 测试的命令非常简单， `rspec` 就可以，他会自动运行 spec 文件夹下所有的 xxx_spec.rb 文件，也可以指定某个文件：

```
rspec spec/models/product_spec.rb
```

也可以只运行某一个测试用例，这需要指定该用例开始的行数：

```
rspec spec/models/product_spec.rb:10
```

也可以运行某一个目录：

```
rspec spec/models/
```

factory_girl_rails 是 factory_girl 的 Rails 包装。factory_girl 可以为我们的测试代码提供模拟的测试数据。

database_cleaner 可以在每一次运行测试的时候，清空测试数据库。我们在 config/database.yml 中，会设置三种运行环境，test 环境要单独设置数据库，也就是因为测试时会反复填入和删除数据。一般，test 使用的是 sqlite 数据库，而 production 使用 mysql、postgresql 等数据库。

我们需要配置下 spec 的运行环境：

```
RSpec.configure do |config|
  config.before(:each) do
    DatabaseCleaner.strategy = :truncation
    DatabaseCleaner.clean
  end
end
```

4.2.5.1 Model 测试

在使用 generator 创建 model 文件的时候，rspec 会自动创建它对应的 spec 文件。我们打开 product_spec.rb 文件：

```
require 'rails_helper'

RSpec.describe Product, type: :model do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

我们为它增加一个测试：

```
RSpec.describe Product, type: :model do
  it "should create a product" do
    tshirt = Product.create(name: "T-Shirt", price: 9.99)

    expect(tshirt.name).to eq("T-Shirt")
    expect(tshirt.price).to eq(9.99)
  end
end
```

运行一下这个测试：

```
rspec spec/models/product_spec.rb
.

Finished in 0.081 seconds (files took 2.37 seconds to load)
1 example, 0 failures
```

这个测试的目的，是确保 create 方法可以为我们创建一个 product 实例。更多 rspec 语法可以查看 rspec 文档，或者《使用 RSpec 测试 Rails 程序》一书。

4.3 模型中的关联关系（Relations）

概要：

本课时讲解 Rails 中 Model 和 Model 间的关联关系。

知识点：

1. belongs_to
2. has_one
3. has_many
4. has_and_belongs_to_many
5. self join

正文

导读

如果你对一对一关系，一对多关系，多对多关系并不十分了解的话，或者你对关系型数据库并不十分了解的话，建议你在阅读下面的内容前，先熟悉一下相关内容。因为我并不想照本宣科的讲解手册。我想讲的，是对它的理解，并且把我们的精力，放到设计我们的商城中。

本章涉及的知识，可以查看 Active Record Associations，或者 ActiveRecord::Associations::ClassMethods。

接下来的内容，希望能帮助你理解模型间的关联关系。

4.3.1 模型间的关系

在前面的章节里，我们为商城设计了界面，并且使用了3个 model:

1. User，网站用户，使用 devise 提供了用户注册，登录功能。
2. Product，商品
3. Variant，商品类型

我们在前面讲解的过程中，已经提到了 Product 和 Variant 的关系。一个 Product 有多个 Variant。现在我们需要增加几个模型，模型是根据功能来的，我们的网店要增加哪些功能呢？

- 当用户购买实物商品的时候，我们是要输入它的收货地址（Address）。
- 当用户选择商品的时候，选择不同的颜色和大小，会有不同的价格（Variant）。
- 我们点击购买，会创建一个购物订单（Order），上面有我们选择的商品，应支付的金额，和订单的状态。
- 查看用户购买的商品类型

在我们的网店里，一个 User 有一个地址，每次购物的时候，会读取这个地址作为送货地址。

一个 Product 有多个 Variant，每个 Variant 保存它的颜色，大小等属性。

一个用户会有多个订单 Order，每个订单会显示购买的商品 Product，以及多条购买记录，每条记录显示购买的 Variant 的每个数量和应付的价格，这里我们使用 LineItem 表示订单的订单项。

4.3.2 外键

两个 model 之间，通过外键进行关联，Rails 中默认的外键名称是所属 model 的 `名称_id`，比如，User 有一条 Address 记录，那么 addresses 表上，需要增加一个数字类型的字段 `user_id`。而 User 的主键通常为 id 字段。有一些遗留的数据库，使用的外键可能不是按照 Rails 默认的格式，所以在声明外键关联时，需要指定 `foreign_key`。

在我们创建 Model 的时候，可以在 generate 命令上增加外键关联，我们现在创建 Address 这个 Model

```
rails g model address user:references state city address address2 zipcode receiver pho
ne
```

在创建的 migration 文件中：

```
create_table :addresses do |t|
  t.references :user, index: true, foreign_key: true
```

自动增加了外键关联，并且将 user_id 加入索引。如果是更改其他数据库，需要在 migration 文件内单独设置索引：

```
add_index "addresses", ["user_id"], name: "index_addresses_on_user_id"
```

模型间的关系，都是通过外键实现的，下面我们详细介绍模型间的关系，并且实现我们商城的 Model。

4.3.3 一对一关系

一对一关系的设定，再一次体现了 Rails 在开发中的便捷：

```
class User < ActiveRecord::Base
  has_one :address
end

class Address < ActiveRecord::Base
  belongs_to :user
end
```

在一对一关系中，`belongs_to :user` 中，`:user` 是单数，`has_one :address` 中，`:address` 也是单数。

我们进入到 console 里来测试一下：

```
user = User.first
user.address
=> nil
```

4.3.3.1 新建子资源

如何为 user 保存 address 呢？

一种是使用 Address 的类方法 `create`：

```
Address.create(user_id: user.id, ...)
```

我们也可以省去 id 的写法，直接写上所属的实例：

```
Address.create(user: user, ...)
```

一种是使用实例方法：

```
address = Address.new
address.user = user
address.save
```

或者：

```
user.address = Address.create( ... )
```

这种方法会产生两句 SQL，先是 insert 一个 address 到数据库，然后更新它的 user_id 为刚才的 user。我们可以换一个方法：

```
user.address = Address.new( ... )
```

它只产生一条 insert SQL，并且会带上 user_id 的值。

在创建关联关系时，还有这样的方法：

```
user.create_address( ... )
user.build_address( ... )
```

build_xxx 相当于 Address.new。create_xxx也会产生两条 SQL，每条 SQL 都包含在一个 transaction 中。

所以我们得出结论：

把一个未保存的实例，赋值给一对一关系时，它会自动保存，并且只有一条 sql 产生。

先 create 一个实例，再把赋值给一对一关系时，是先保存，再更新，产生两条 sql。

4.3.3.2 保存子资源

当我们编写表单的时候，一个表单针对的是一个资源。当这个资源拥有（has_one 或 has_many）子资源时，我们可以在提交表单的时候，将它拥有的资源也保存到数据库中。

这时，我们需要在 User中，做一个声明：

```
class User < ActiveRecord::Base
  has_one :address
  accepts_nested_attributes_for :address
end
```

`accepts_nested_attributes_for` 会为 User 增加一个新的方法 `address_attributes=(attributes)` ，这样，在创建 User 的 时候：

```
user_hash = { email: "test@123.com", password: "123456", password_confirmation: "12345
6", address_attributes: { receiver: "Some One", state: "Beijing", city: "Beijing", pho
ne: "123456"} }
u = User.create(user_hash)
u.address
```

只要保存 User 的时候，传递入 Address 的参数，就可以把关联的 address 一并保存到数据库中了。

更新记录的时候，也可以使用同样的方法：

```
user_hash = { email: "changed@123.com", address_attributes: { receiver: "Other One" }
}
user.update(user_hash)
```

但是，这里要注意，上面的方法会把之前旧记录的 user_id 设为 nil，然后插入一条新的记录。这并不能真正起到更新的作用，除非所有属性都重新复制，不然，新的 address 记录只有 receiver 这个值。

我们在 accepts_nested_attributes_for 后增加一个参数：

```
accepts_nested_attributes_for :address, update_only: true
```

这样，update 时候会更新已有的记录。

如果我们不能增加 `update_only` 属性，为了避免创建无用的记录，需要在 hash 里指定子资源的 id：

```
user_hash = { email: "changed@123.com", address_attributes: { id: 1, receiver: "Other
One" } }
user.update(user_hash)
```

4.3.3.3 使用表单保存子资源

`accepts_nested_attributes_for` 方法，在 Form 中有其对应的方法：

```
<%= f.fields_for :address do |address_form| %>
  <%= address_form.hidden_field :id unless resource.new_record? %>
  <div class="form-group">
    <%= address_form.label :state, class: "control-label" %><br />
    <%= address_form.text_field :state, class: "form-control" %>
  </div>
  ...
<% end %>
```

打开 代码，在编辑一个用户的时候，我为它增加了一个 `f.fields_for` 的子表单，对应了子资源的属性。

我想，这段代码这并不难理解，不过我们用了 Devise 这个 gem，还需要做一点额外的处理。

打开 application_controller.rb，我们需要让 devise 支持传进来新增的参数：

```
class ApplicationController < ActionController::Base
  before_action :configure_permitted_parameters, if: :devise_controller?

  protected

  def configure_permitted_parameters
    devise_parameter_sanitizer.for(:sign_up) { |u| u.permit(:email, :password, :passwo
rd_confirmation, :address_attributes) }
    devise_parameter_sanitizer.for(:account_update) { |u| u.permit(:email, :password,
:password_confirmation, :current_password, address_attributes: [:state, :city, :addres
s, :address2, :zipcode, :receiver, :phone] ) }
  end
end
```

在我们注册账号的时候，并没有创建 address ，但是在编辑的时候，因为它是 nil，所以不会显示这个子表单，所以我们需要在编辑的时候创建一个空的 address：

`views/devise/registrations/edit.html.erb`

```
<%= form_for(resource, as: resource_name, url: registration_path(resource_name), html:
 { method: :put }) do |f| %>
  <% resource.build_address if resource.address.nil? %>
  ...
```

当然，我们也可以在注册的时候提供地址表单，大家不妨一试。

4.3.3.4 删除关联的子资源

在上一节里，我们介绍了 delete 和 destroy 方法，我们可以使用这两个方法把关联的 address 删除掉：

```
u.address.delete
  SQL (10.0ms)  DELETE FROM "addresses" WHERE "addresses"."id" = ?  [["id", 2]]
```

或者：

```
u.address.destroy
   (0.1ms)  begin transaction
  SQL (0.7ms)  DELETE FROM "addresses" WHERE "addresses"."id" = ?  [["id", 3]]
   (9.2ms)  commit transaction
```

两者的区别在上一节介绍过，我们注意到，delete 直接发送数据库删除命令，而 destroy 会将删除命令放置到一个 sql 的事物中，因为它会触发模型中的回调，如果回调抛出异常，删除动作会失败。

4.3.3.5 删除自身同时删除关联的子资源

在删除某个资源的时候，我们想把它拥有的资源一并删除，这时，我们需要给 has_one 方法，增加一个参数：

```
has_one :address, dependent: :destroy
```

dependent 可以接收五个参数：

参数	含义
:destroy	删除拥有的资源
:delete	直接发送删除命令，不会执行回调
:nullify	将拥有的资源外键设为 null
:restrict_with_exception	如果拥有资源，会抛出异常，也就是说，当它 has_one 为 nil 的时候，才能正常删除它自己
:restrict_with_error	如有拥有资源，会增加一个 errors 信息。

在 belongs_to 上，也可以设置 dependent，但它只有两个参数：

参数	含义
:destroy	删除它所属的资源
:delete	删除它所属的资源，直接发送删除命令，不会执行回调

两种设定，出发角度是不同的，不过，删除本身的同时删除上层资源是比较危险的，需谨慎。

4.3.3.6 失去关联关系的子资源

如果在 has_one 中设置了 `dependent: :destroy` 或 `dependent: :delete`，当子资源失去该关联关系时，它也会被删除。

```
user.address = nil
```

如果不设置，一个子资源失去关系时，外键设置为 null。

4.3.3.7 子资源维护

当一个子资源失去关联关系，和它在关联关系中被删除，是一样的。我们在设计时，应尽量避免产生孤立的记录，这些记录外键为 null，或者所属的资源已经被删除，他们是无意义的存在。

4.3.4 一对多关系

在电商系统里，一个用户是有多个订单（Order）的，User 中使用的是 has_many 方法：

```
class User < ActiveRecord::Base
  has_many :orders
end
```

除了名称变为复数形式，返回的结果是数组，其他情形和“一对一”是一样的。

我们使用 generate 创建 Order：

```
rails g model order user:references number payment_state shipment_state
```

number 是订单的唯一编号，payment_state 是付款状态，shipment_state 是发货状态。

payment_state 的状态顺序是：pending（等待支付），paid（已支付）。

shipment_state 的状态顺序是：pending（等待发货），shipped（已发货）。

这两种状态，我们只做简单的设计，实际中要复杂得多。

开源电商程序 spree 是一套很好的在线交易程序，因为其开源，其中的概念和定义对开发电商程序有很好的启发。它的源代码在 这里，目前是最新版本是 3.0.2.beta。

4.3.4.1 添加子资源

一对多关系返回的，是 CollectionProxy 实例。

当添加一对多关系时，可以很“形象”的使用：

```
product.variants << Variant.new
product.variants << [Variant.new, Variant.new]
```

执行 `<<` 的时候，variant 的 product_id 会自动保存为 product.id。

如果 variant 是一个未保存到数据库的实例，<< 执行的时候会自动将它保存，并且赋予它 product_id 值。这是一步完成的，只有一条 SQL。

但是，如果是下面的情形：

```
product.variants << Variant.create
```

会把 variant 先保存到数据库，然后再更新它的 product_id 字段，这会产生两条 SQL。

这里也可以使用 build 方法，和上面“一对一关系”不同的是，它需要在 collection 上执行：

```
variant = product.variants.build( ... )
variant.save
```

build 返回的是一个未保存的实例。查看 `product.variants` ，会看到它包含了一个未保存的 variant（ID 为 nil）。

另一种情形：

```
product.variants.build( ... )
product.save
```

当这个 product.save 的时候，这个 variant 也会保存到数据库中。

4.3.4.2 删除子资源

删除资源的时候，可以使用几个方法：

```
product.variants.delete(...)
product.variants.destroy(...)
product.variants.clear
```

delete 不会真正删除掉资源，而是把它的外键（product_id）设为 nil，而 destroy 会真正的删除掉它并出发回调。

他们都可以传递进一个实例，或者实例的集合，而并不管这个实例是否真的属于它。

```
product.variants.delete(Variant.find(1))
product.variants.delete(Variant.find(1,2,3))
```

这样是不是太霸道了？所以，建议用第三个方法更稳妥些。clear 方法会把外键置为 nil。

如果再 has_many 上声明了 `dependent: :destroy` ，会用 destroy 方式把它们删除（有回调）。如果声明的是 `dependent: :delete_all` ，会用 delete 方法（跳过回调）。这和一对一中描述是一致的。

注意：

has_many 和 has_one 上的 dependent 选项，适用以下两种情形：

- 删除自身时，如何处理子资源
- 当子资源失去该关联关系时，如何处理该子资源

我们来看下一节。

4.3.4.3 更改子资源

当改动关系的时候，可以直接使用 `=` ，假设我们有 ID 为 1，2，3，4 的 Variant：

```
product.variants = Variant.find(1,2)
```

这时会自动把 ID:1，ID:2 的 product_id 外键设为 null。

再次选择 ID:3，ID:4 的 variant：

```
product.variants = Variant.find(3,4)
```

会自动把 ID:3，ID:4 的 product_id 外键设置为 product.id。

如果在 has_many 设置了 `dependent: :destroy` ，当 UD:1 和 ID:2 失去关联的时候，会把它们从数据库中删除掉。这与 has_one 中的 dependent 选项是一样的。详见本章前面 `4.3.3.4 删除自身同时删除关联的子资源` 。

4.3.4.4 counter_cache

“一对多”关系中， `belongs_to` 方法可以增加 counter_cache 属性：

```
class Order < ActiveRecord::Base
  belongs_to :user, counter_cache: true
end
```

这时，我们需要给 users 表增加一个字段：orders_count，当我们把一个 order 保存到一对多的关系中时，orders_count 会自动 +1，当把一个资源从关系中删除，该字段会 -1。如此我们不必去增加计算一个 user 有多少个 orders，只需要读该字段就可以了。

向 Users 表添加 orders_count 字段：

```
rails g migration add_orders_count_to_users orders_count:integer
```

4.3.4.5 多态

当一个资源可能属于多种资源时，可以用到多态。举个栗子：

商品可以评论，文章可以评论，而评论 model 对任何一个资源都是一样的功能，所以，评论在 belongs_to 的后面，增加：

```
class Comment < ActiveRecord::Base
    belongs_to :commentable, polymorphic: true
end
```

Comment 的迁移文件，也相应的增加设定：

```
t.references :commentable, polymorphic: true, index: true
```

如果是手动添加字段，需要这样来写：

```
t.string :commentable_type
t.integer :commentable_id
```

说明，查找一个多态资源时，是根据拥有者的类型（type，一般是它的类名称）和 ID 进行匹配的。

拥有评论的 model，也需要改动下：

```
class Product < ActiveRecord::Base
  has_many :commentable, as: :commentable
end

class Topic < ActiveRecord::Base
  has_many :commentable, as: :commentable
end
```

多态并不局限于一对多关系，一对一也同样适用。

4.3.5 中间模型和中间表

has_one 和 has_many，是两个 model 间的操作。我们可以增加一个中间模型，描述之前两个 model间的关系。

4.3.5.1 中间模型

我们先创建订单项（LineItem）这个 model，它属于一个订单，也属于一个商品类型（Variant）。

```
rails g model line_item order:references variant:references quantity:integer
```

对于一个订单，我们有多个订单项，对于一个订单项，会关联购买的具体商品类型，那么，一个订单拥有的商品类型，就可以通过 through 查找到。

```
class Order < ActiveRecord::Base
  belongs_to :user, counter_cache: true
  has_many :line_items
  has_many :variants, through: :line_items
end
```

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :variant
end
```

我们进到终端里进行查找：

```
order = Order.first
order.variants
=> SELECT "variants".* FROM "variants" INNER JOIN "line_items" ON "variants"."id" = "l
ine_items"."variant_id" WHERE "line_items"."order_id" = ?  [["order_id", 1]]
 => #<ActiveRecord::Associations::CollectionProxy []>
```

可以看到，through 为使用了 `inner join` 的 sql 语法。

LineItem 是两个模型，Order 和 Variant 的中间模型，它表示订单中的每一项。但是，中间模型不一定要使用两个 `belongs_to` 连接两边的模型，比如：

```
class User < ActiveRecord::Base
  has_many :orders
  has_many :line_items, through: :orders
end
```

进到终端，我们查看一个用户有哪些订单项：

```
user = User.first
user.line_items
=> SELECT "line_items".* FROM "line_items" INNER JOIN "orders" ON "line_items"."order_
id" = "orders"."id" WHERE "orders"."user_id" = ?  [["user_id", 1]]
```

从左边可以查到右边资源，那么，可以通过中间表，从右边查找左边资源么？

我们给 Variant 增加关联：

```
class Variant < ActiveRecord::Base
  belongs_to :product
  has_many :line_item
  has_many :orders, through: :line_item
end
```

进入终端：

```
v = Variant.last
v.orders
=> SELECT "orders".* FROM "orders" INNER JOIN "line_items" ON "orders"."id" = "line_it
ems"."order_id" WHERE "line_items"."variant_id" = ?  [["variant_id", 2]]
```

因为中间表 LineItem 拥有两边的外键，所以可以查找 variant 的 orders。但是 orders 上没有 line_item_id 字段，因为这不符合我们的业务逻辑，所以无法查找 line_item.user。如果需要查找，可以给 line_item 上增加 user_id 字段。

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :variant
  belongs_to :user
end
```

4.3.5.2 中间表

中间模型的作用，除了连接两端模型外，更重要的是，它保存了业务中属于中间模型的数据，比如，订单项中的 quantity 字段。如果模型不必或者没有这种字段，可以不用增加 model，而直接使用中间表。

我们有一个功能：保存用户购买的商品类型。这时可以使用中间表，保存购买关系。

中间表具有两端模型的外键。两端模型使用 `has_and_belongs_to_many` 方法（简写：HABTM）。

在创建中间表的时候，也可以使用 migration，如果在表名中包含 `JoinTable` 字样，会自动创建中间表：

```
rails g migration CreateJoinTable users variants:uniq
```

运行 `rake db:migrate`，查看 schema.rb：

```
create_table "users_variants", id: false, force: :cascade do |t|
  t.integer "user_id",    null: false
  t.integer "variant_id", null: false
end

add_index "users_variants", ["variant_id", "user_id"], name: "index_users_variants_on_
variant_id_and_user_id", unique: true
```

调整一下 User 和 Variant model：

```
class User < ActiveRecord::Base
  ...
  has_and_belongs_to_many :variants
end

class Variant < ActiveRecord::Base
  ...
  has_and_belongs_to_many :users
end
```

在终端里测试：

```
user.variants
=> SELECT "variants".* FROM "variants" INNER JOIN "users_variants" ON "variants"."id"
= "users_variants"."variant_id" WHERE "users_variants"."user_id" = ?  [["user_id", 1]]

variant.users

=> SELECT "users".* FROM "users" INNER JOIN "users_variants" ON "users"."id" = "users_
variants"."user_id" WHERE "users_variants"."variant_id" = ?  [["variant_id", 2]]
```

利用中间表，实现了多对多关系。

4.3.5.3 多对多关系

查看一个用户购买了哪些商品类型，和查看一个商品类型被哪些用户购买，这就是多对多关系。

保存和删除多对多关系，和一对多关系的操作是一样的。因为我们在创建 migration 时，增加了索引唯一校验，在操作时要做好异常处理，或者保存前进行判断。

```
user.variants << variant
user.variants << variant
=> SQLite3::ConstraintException: columns variant_id, user_id are not unique: ...
```

4.3.5.4 inner join

ActiveRecord 在查询关联关系时，使用的是 inner join 查询，我们可以单独使用 `join` 方法，实现该查询。

比如，一个简单的 join 查询：

```
% Order.joins(:line_items)
=> SELECT "orders".* FROM "orders" INNER JOIN "line_items" ON "line_items"."order_id"
= "orders"."id"
```

也可以查询多个关联的：

```
% Order.joins(:line_items, :user)
=> SELECT "orders".* FROM "orders" INNER JOIN "line_items" ON "line_items"."order_id"
= "orders"."id" INNER JOIN "users" ON "users"."id" = "orders"."user_id"
```

或者嵌套关联：

```
% Order.joins(line_items: [:variant])
=> SELECT "orders".* FROM "orders" INNER JOIN "line_items" ON "line_items"."order_id"
= "orders"."id" INNER JOIN "variants" ON "variants"."id" = "line_items"."variant_id"
```

但是，在一些更复杂的查询中，我们需要改变 `inner join` 查询为 `left join` 或 `right join`：

```
User.select("users.*, orders.*").joins("LEFT JOIN `orders` ON orders.user_id = users.i
d")
```

这时返回的是全部用户，即便它没有订单。这在生成一些报表时是有用的。

4.3.6 自连接

在设计模型的时候，一个模型即可以是 Catalog（类别），也可以是 Subcatalog（子类别），我们为网店添加 `类别` Model：

```
rails g model catalog parent_catalog:references name parent:boolean
```

看一下 catalog.rb：

```
class Catalog < ActiveRecord::Base
  has_many :subcatalogs, class_name: "Catalog", foreign_key: "parent_catalog_id"
  belongs_to :parent_catalog, class_name: "Catalog"
  has_many :products
end
```

这样，我们可以实现分类，也可以吧商品加入到某个分类中。

4.3.7 双向关联

我们查找关联关系的时候，是可以在两边同时查找，比如：

```
class User < ActiveRecord::Base
  has_one :address
end

class Address < ActiveRecord::Base
  belongs_to :user
end
```

我们可以 `user.address` ，也可以 `address.user` ，这叫做 Bi-directional，双向关联。（和它相反，Uni-directional，单向关联）

但是，这在我们的内存查找中，会引起问题：

```
u = User.first
a = u.address
u.email == a.user.email
=> true
u.email = "a@1.com"
u.email == a.user.email
=> false
```

原因是：

```
u.object_id
 => 70241969456560
a.user.object_id
 => 70241969637580
```

两个类并不是在内存中指向同一个地址，他们是不同的两个类。

为了避免这个问题，我们需要使用 inverse_of：

```
class User < ActiveRecord::Base
  has_one :address, inverse_of: :user
end

class Address < ActiveRecord::Base
  belongs_to :user, inverse_of: :address
end
```

当 model 的关联关系上，已经有 polymorphic，through，as 时，可以不用加 inverse_of，它自然会指向同一个 object，大家可以使用 user 和 order 之间的关联验证。对于 user 和 address 之间，还是应该加上 inverse_of 选项。

4.3.8 Rspec测试

关联关系的测试，可以使用 shoulda-matchers 这个 gem。它为 Rails 的模型间关联提供了方便的测试方法。

比如：

```
RSpec.describe User, type: :model do
  it { should have_many(:orders) }
end

RSpec.describe Order, type: :model do
  it { should belong_to(:user) }
end
```

更多模型间关联关系测试的方法，可以查看 ActiveRecord matchers

4.4 模型中的校验（Validates）

概要：

本课时讲解 Model 中的属性校验方法，以及在页面上显示校验失败信息。

知识点：

1. validates 方法
2. errors
3. helpers
4. I18n
5. Rspec

正文

4.4.1 数据校验

我们将数据保存到数据库的时候，可以有两种数据校验，一种是在数据库中设定验证规则，一种是在程序中进行校验。

Rails 为我们提供了方便的属性校验。在 [4.2.1 两个 Gem] 一节，我们介绍了ActiveRecord 中包含的两个 Gem，在数据查询和关联关系中，我们主要使用的是 arel。数据校验时，我们使用的是 ActiveModel。

4.4.2 校验方法

4.4.2.1 常用的校验方法

方法	含义	例子
acceptance	必须接受选项，比如注册条款（必须同意）	validates :terms_of_service, acceptance: true
validates_associated	校验关联资源，仅在关联的一端使用即可，避免循环校验	[1]
confirmation	填写确认	validates :email, confirmation: true
exclusion	排除内容，如某些保留关键词不允许注册使用	validates :subdomain, exclusion: { in: %w(www us ca jp), message: "%{value} is reserved." }
format	格式化，如邮件格式	validates :legacy_code, format: { with: /\A[a-zA-Z]+\z/, message: "only allows letters" }
inclusion	包含内容，如特定的输入内容	validates :size, inclusion: { in: %w(small medium large), message: "%{value} is not a valid size" }
length	内容长度	validates :name, length: { minimum: 2 } [2]
numericality	仅数字	validates :points, numericality: true
presence	必填，使用 `blank?` 方法判断	validates :name, :login, :email, presence: true [3] [4]
absence	必空，使用 `present?` 判断	[5]
uniqueness	唯一	validates :email, uniqueness: true [6]

注解：

[1]

```
class Library < ActiveRecord::Base
  has_many :books
  validates_associated :books
end
```

[2] 有其他几个选项：

:minimum，最短长度 :maximum，最大长度 :in/:within，在某范围 :is，指定长度

[3] 也可以应用在关联关系上，如：

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order, presence: true
end
```

为了保持内存中引用相同地址，需要在 Order 上使用 inverse_of：

```
class Order < ActiveRecord::Base
  has_many :line_items, inverse_of: :order
end
```

[4] 进入 console，做个试验：

```
false.blank?
  => true
true.blank?
  => false
```

所以，使用 presence 判断 true/false 属性时，需要这样使用：

```
validates :boolean_field_name, presence: true
validates :boolean_field_name, inclusion: { in: [true, false] }
validates :boolean_field_name, exclusion: { in: [nil] }
```

[5] 和 presence 一样，需要使用 inverse_of 限定关联关系，并且在判断 true/false 时：

```
validates :boolean_field_name, absence: true
validates :boolean_field_name, exclusion: { in: [true, false] }
```

[6] uniqueness 有两个重要的选项。

scope，比如：

```
validates :number, uniqueness: { scope: : company_id }
```

保存到数据库前，uniqueness 会先检索数据库是否已经存在该字段的值，scope 可以使检索时附带一个字段，比如：不同的公司，可以有相同的订单号，而同公司订单号必须唯一。

```
validates :name, uniqueness: { case_sensitive: false }
```

默认是 true，区分大小写。改为 false，可不区分大小写。

4.4.2.2 校验方法中的选项

在检验方法 validates 中，可以使用几个选项：

选项	含义	例子
allow_nil	是否允许为 nil	validates :size, allow_nil: true
allow_blank	是否允许为 blank?，为 false 时，不可填写 `""`，`false`，`nil`	validates :title, allow_blank: true
message	自定义错误信息	validates :subdomain, exclusion: { in: %w(www us ca jp), message: "%{value} 为保留关键词" }
on	选择在 create 或 update 上使用校验	validates :email, uniqueness: true, on: :create
strict	校验失败时抛出异常，或自定异常类	validates :name, presence: { strict: true } [1]

注解

[1]

自定义异常类

```
class Person < ActiveRecord::Base
  validates :token, presence: true, uniqueness: true, strict: TokenGenerationException
end

Person.new.valid?
=> TokenGenerationException: Token can't be blank
```

4.4.3 触发校验方法

在将数据保存到数据库的时候，有些方法，会触发校验，有些则直接发送数据库 sql 命令，不触发校验。

4.4.3.1 触发校验的方法

- create
- create!
- save
- save!
- update
- update!

`!` 结尾的方法，在校验失败时，会抛出异常。`save(validate: false)` 可以跳过 save 方法的校验。

4.4.3.2 不触发校验的方法

- decrement!
- decrement_counter
- increment!
- increment_counter
- toggle!
- touch
- update_all
- update_attribute
- update_column
- update_columns
- update_counters

4.4.3.2 有条件的校验

我们可以在校验中增加 `:if` 或 `:unless` 条件判断。

```
class Order < ActiveRecord::Base
  validates :card_number, presence: true, if: :paid_with_card?
  def paid_with_card?
    payment_type == "card"
  end
end
```

这里使用的是方法判断，也可以直接使用字符串，比如：

```
class Person < ActiveRecord::Base
  validates :surname, presence: true, if: "name.nil?"
end
```

或者一个代码块：

```
class Account < ActiveRecord::Base
  validates :password, confirmation: true, unless: Proc.new { |a| a.password.blank? }
end
```

4.4.3.3 `valid?` 方法

`valid?` 和 `invalid?` 方法会触发校验。校验成功时返回 true，失败时，返回 false，并且将校验信息放入 errors 类。访问 `order.errors`，返回的是 `ActiveModel::Errors` 实例，它的代码在 这里。

4.4.4 Errors 对象

校验失败时，model.errors 会保存入校验的属性和失败原因。我们可以通过几个方法，从 errors 实例中拿到具体的信息。

```
% model.errors.messages
=> {:number=>["must be blank"]}
```

`messages` 方法返回的是 hash 结构的信息，key 是校验的属性。

```
% model.errors.full_messages
=> ["Number can't be blank", ...]
```

`full_messages` 方法返回 Array 结构的完整错误信息。这在资源编辑的 form 页面，可以整体输出错误信息，不过它没有具体到某个属性上。对于某个属性，我们可以使用 `errors[:number]` 来读取：

```
% order.errors[:number]
=> ["can't be blank"]
```

在某些时候，我们需要添加自己的信息，可以使用：

```
order.errors.add(:number, "订单号不能含有 !@#%*()_-+= 等字符")
```

如果添加的信息，并不一定是某个具体属性，可以添加到errors 的 base 中：

```
order.errors.add(:base, "订单格式不正确")
```

`order.errors.clear`，可以清理掉所有信息.

4.4.5 使用中文的校验信息

我们已经注意到了，目前所有的校验信息都是英文的，虽然可以在自定义信息里写入中文（Not Rails Style），但是我们可以利用 Rails 提供的 I18n gem，实现文本内容的汉化。这包括异常信息。

我们先修改一下 I18n 文件加载地址，在 application.rb 文件里，我们找到这一段：

```
config.i18n.load_path += Dir[Rails.root.join('config', 'locales', '**/*.{rb,yml}').to_
s]
config.i18n.default_locale = :"zh-CN"
```

这样会加载我们在 config/locales 中的全部语言包文件（注意，这里使用的是 `**/*.{rb,yml}` ）。

我们创建语言包，为了便于维护，我在这里做了细分，大家可以在 这里 查看。

进到终端里，测试下：

```
% product = Product.new
% product.valid?
=> false
% product.errors.full_messages
=> ["名称不能为空字符"]
```

在后面的章节里，会专门讲解 I18n 的问题，如果不像本例子中自己添加语言包，也可以安装 rails-i18n 这个 gem 来解决问题。

4.4.5.1 页面中显示错误信息

为了让页面集中的显示错误信息，我们在 form 中使用了局部模板，把校验失败的内容显示在输入框的顶部。

```
<% if @product.errors.any? %>
  <div id="error_expl" class="panel panel-danger">
    <div class="panel-heading">
      <h3 class="panel-title"><%= pluralize(@product.errors.count, "error") %> prohibi
ted this product from being saved:</h3>
    </div>
    <div class="panel-body">
      <ul>
      <% @product.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  </div>
<% end %>
```

`full_messages` 返回 Array 的校验信息，我们只需循环显示即可。如果想在输入框旁边显示信息，可以单独读取该属性，比如 `@product.errors[:name]` ，可以放到一个 jquery 的 tooltip 中。

不过，这种信息是要提交到服务器端处理后，才能显示出来的。为了在页面端就显示校验，我们还是需要 jQuery 插件的。

4.4.5.2 jQuery 校验

Form 校验的时候，有两个插件较常用。

http://jqueryvalidation.org/ 是较常用的一个，也很简单，但是需要在页面上显示中文，还需要它的中文插件。

<%= javascript_include_tag 'spree/jquery.validate/localization/messages_zh' %>

中文语言包的源码在[这里] (https://github.com/jzaefferer/jquery-validation/blob/master/src/localization/messages_zh.js)。

如果不需要校验具体信息，因为我们已经使用了 bootstrap 这个前端框架，所以我们可以使用它的表单校验：http://bootstrapvalidator.com

它会按照 bootstrap 的方式，将输入框加上图标，使校验更加直观。当然，你还可以读取具体的属性信息，放到 bootstrap 的 tooltip 里。

4.4.6 Rspec

和上一张的关联关系一样，shoulda-matchers 也提供了方便的校验测试框架。

```
describe Product do
  it { should validate_presence_of(:name) }
end
```

现在我们给 Model 增加了越来越多的内容，为了方便找到方法，我们可以对代码进行一个简单的分割，这样就不会在测试和对应的业务代码间切换浪费时间了。

```
# extends ...................................................................
# includes ..................................................................
# security ..................................................................
# relationships .............................................................
# validations ...............................................................
# callbacks .................................................................
# scopes ....................................................................
# additional config .........................................................
# class methods .............................................................
# public instance methods ...................................................
# protected instance methods ................................................
# private instance methods ..................................................
```

4.5 模型中的回调（Callback）

概要：

本课时将讲解 ActiveRecord 中常用的回调方法。

知识点：

1. ActiveModel 中的回调
2. ActiveRecord 中的回调
3. 编写回调
4. 触发回调
5. 使用回调计算库存

正文

4.5.1 ActiveModel 中的回调

ActiveModel 提供了多个实用的功能，它可以让一个普通的类，具备如属性校验，回调，显示字段 I18n 值等众多功能。

比如，我们可以为 Person 类增加了一个回调方法：

```
class Person
  extend ActiveModel::Callbacks
  define_model_callbacks :create
end
```

所谓回调，是指在某个方法前（before）、后（after）、前后（around），执行某个方法。上面的例子里，Person 拥有了三个标准的回调方法：before_create、after_create、around_create。

我们还需要为这个回调方法增加逻辑代码：

```
class Person
  extend ActiveModel::Callbacks
  define_model_callbacks :create

  # 定义 create 方法代码
  def create
    run_callbacks :create do
      puts "I am in create method."
    end
  end

  # 开始定义回调
  before_create :action_before_create
  def action_before_create
    puts "I am in before action of create."
  end

  after_create :action_after_create
  def action_after_create
    puts "I am in after action of create."
  end

  around_create :action_around_create
  def action_around_create
    puts "I am in around action of create."
    yield
    puts "I am in around action of create."
  end
end
```

进入到 Rails 的终端里，我们测试下这个类：

```
% rails c
> person = Person.new
> person.create
I am in before action of create.
I am in around action of create.
I am in create method.
I am in around action of create.
I am in after action of create.
```

在 ActionModel 中有许多的 Ruby 元编程知识，如果你感兴趣，可以读一读《Ruby 元编程（第二版）》这本书。

ActiveRecord 中的 回调 将常用的 `find`，`create`，`update`，`destroy` 等方法进行包装。

Rails 在 controller 也有回调，我们下一章会介绍。

4.5.2 ActiveRecord 中的回调

我们在 Rails 中使用的 Model 回调，是通过调用 ActiveRecord 中定义的 `实例方法` 来实现的，比如 `before_validation` 方法，实现了在 `validate` 方法前的回调。

所谓 `回调`，就是在目标方法上，再执行其他的方法代码。

ActiveRecord 提供了众多回调方法，包含了一个 model 实例在数据库操作中的各个时期。按照数据库操作的不同，可以将它们划分为五种情形的回调方法。

第一种，创建对象时的回调。

- before_validation
- after_validation
- before_save
- around_save
- before_create
- around_create
- after_create
- after_save
- after_commit/after_rollback

第二种，更新对象时的回调。

- before_validation
- after_validation
- before_save
- around_save
- before_update
- around_update
- after_update
- after_save
- after_commit/after_rollback

第三种，删除对象时的回调。

- before_destroy
- around_destroy
- after_destroy
- after_commit/after_rollback

第四种，初始化和查找时的回调。

- after_find

- after_initialize

after_initialize 会在一个实例使用 new 创建，或从数据库读取时触发。这样避免直接覆写实例的 initialize 方法。

当从数据库读取数据时，会触发 after_find 回调：

- all
- first
- find
- find_by
- find*by**
- find*by**!
- find_by_sql
- last

after_find 执行优先于 after_initialize。

第五种，**touch** 回调。

- after_touch

执行实例的 `touch` 方法触发该回调。

回调执行顺序

我们观察一下以上每个回调的执行的顺序，这里做一个简单的例子：

```
class Product < ActiveRecord::Base
  before_validation do
    puts "before_validation"
  end

  after_validation do
    puts "after_validation"
  end

  before_save do
    puts "before_save"
  end

  around_save :test_around_save
  def test_around_save
    puts "begin around_save"
    yield
    puts "end around_save"
  end

  before_create do
    puts "before_create"
  end

  around_create :test_around_create
  def test_around_create
    puts "begin around_create"
    yield
    puts "end around_create"
  end

  after_create do
    puts "after_create"
  end

  after_save do
    puts "after_save"
  end

  after_commit do
    puts "after_commit"
  end

  after_rollback do
    puts "after_rollback"
  end
end
```

进入终端试验下：

```
product = Product.new(name: "TTT")
product.save
 (0.1ms)  begin transaction
before_validation
after_validation
before_save
begin around_save
before_create
begin around_create
  SQL (0.6ms)  INSERT INTO "products" ("name", "created_at", "updated_at") VALUES (?,
?, ?)  [["name", "TTT"], ["created_at", "2015-06-16 02:49:20.871384"], ["updated_at",
"2015-06-16 02:49:20.871384"]]
end around_create
after_create
end around_save
after_save
 (0.7ms)  commit transaction
after_commit
=> true
```

可以看到，create 回调是最接近 sql 执行的，并且 validation、save、create 回调被包含在一个 transaction 事务中，最后，是 after_commit 回调。

我们在设计逻辑的过程中，需要了解它执行的顺序。当需要在回调中操作保存到数据库后的实例，需要把代码放到 在 `after_commit` 中。

4.5.3 编写回调

上面列出的，是回调的方法名，我们还需要编写具体的回调代码。

4.5.3.1 符号和方法

```
class Topic < ActiveRecord::Base
  before_destroy :delete_parents [1]

  private [2]
    def delete_parents [3]
      self.class.delete_all "parent_id = #{id}"
    end
end
```

[1] 用符号定义回调执行的方法名称 [2] private 或 protected 方法均可作为回调执行方法 [3] 执行的方法名，和定义的符号一致

对于 `round_` 回调，我们需要在方法中使用 `yield`，上面的例子已经看到：

```
around_create :test_around_create
def test_around_create
  puts "begin around_create"
  yield
  puts "end around_create"
end
```

4.5.3.2 代码块（Block）

```
before_create do
  self.name = login.capitalize if name.blank?
end
```

回调执行时，self 指的是它本身。在注册的时候，我们可能不需要填写 name，而要填写 login，所以默认把 name 改成 login 的首字母大写形式。

上面例子也可以改写成：

```
before_create { |record|
  record.name = record.login.capitalize if record.name.blank?
}
```

4.5.3.3 在特定方法上使用回调

在一些注册和修改的逻辑中，注册时默认填写的数据，在修改时不做处理，所以回调方法只在 create 上生效，下面的例子就是这种情形：

```
before_validation(on: :create) do
  self.number = number.gsub(/[^0-9]/, "")
end
```

或者：

```
before_validation :normalize_name, on: :create
```

4.5.3.4 有条件的回调

和校验一样，回调也可以增加 if 或 unless 判断：

```
before_save :normalize_card_number, if: :paid_with_card?
```

4.5.3.5 字符串形式的回调

```
class Topic < ActiveRecord::Base
  before_destroy 'self.class.delete_all "parent_id = #{id}"'
end
```

`before_destroy` 既可以接受符号定义的方法名，也可以接受字符串。这种方式要被废弃掉了。

4.5.3.6 回调的继承

一个类集成自另一个类，也会继承它的回调，比如：

```
class Topic < ActiveRecord::Base
  before_destroy :destroy_author
end

class Reply < Topic
  before_destroy :destroy_readers
end
```

在执行 `Reply#destroy` 的时候，两个回调都会被执行，为了避免这种情况，可以覆写 `before_destroy`：

```
class Reply < Topic
  def before_destroy() destroy_readers end
end
```

但是，这是非常不好的解决方案！这个代码只是一个例子，来自 这里。

回调虽然可以解决问题，但是它功能太过强大，当项目代码变得复杂，回调的维护会造成很大的技术难度。建议使用回调解决小问题，过多的业务逻辑应该单独处理，或者使用单独的回调类。

4.5.3.6 单独的回调类

我们可以用一个类作为 `回调类` ，使用它的的实例方法实现回调逻辑：

```
class BankAccount < ActiveRecord::Base
  before_save EncryptionWrapper.new
end

class EncryptionWrapper
  def before_save(record) [1]
    record.credit_card_number = encrypt(record.credit_card_number)
  end
end
```

[1] 该方法仅能接受一个参数，为该 model 实例。

还可以使用 `回调类` 的类方法，来定义回调逻辑：

```
class PictureFileCallbacks
  def self.after_destroy(picture_file)
    ...
  end
end
```

在使用上：

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks
end
```

使用单独的回调类，可以方便我们维护回调代码，但是使用起来也需慎重考虑，不要增加后期的维护难度。

4.5.4 触发回调

在我们前面讲解中，更新一个记录时，destroy 方法会触发校验和回调，而 delete 方法不会。在这里详细的列出，ActiveRecord 方法中，哪些会触发回调，哪些不会。

触发回调：

- create
- create!
- decrement!
- destroy
- destroy!
- destroy_all
- increment!
- save

- save!
- save(validate: false)
- toggle!
- update_attribute
- update
- update!
- valid?

不触发回调：

- decrement
- decrement_counter
- delete
- delete_all
- increment
- increment_counter
- toggle
- touch
- update_column
- update_columns
- update_all
- update_counters

4.5.5 回调的失败

所有的回调，在动作执行的过程中，是顺序触发的。在 `before_xxx` 回调中，如果返回 `false` ，这个回调过程会被终止，并且触发数据库事务的 `rollback` ，以及 `after_rollback` 回调。

但是，对于 `after_xxx` 回调，就只能用 `raise` 抛出异常的方式，来终止它。这里抛出的异常必须是 `ActiveRecord::Rollback` 。我们修改下 `after_create` 回调：

```
after_create do
  puts "after_create"
  raise ActiveRecord::Rollback
end
```

在终端里：

```
> Product.create
   (0.1ms)  begin transaction
before_validation
after_validation
before_save
begin around_save
before_create
begin around_create
  SQL (0.4ms)  INSERT INTO "products" ("created_at", "updated_at") VALUES (?, ?)  [["c
reated_at", "2015-08-03 15:30:20.552783"], ["updated_at", "2015-08-03 15:30:20.552783"
]]
end around_create
after_create
   (8.5ms)  rollback transaction
after_rollback
 => #<Product id: nil, name: nil, price: nil, description: nil, created_at: "2015-08-0
3 15:30:20", updated_at: "2015-08-03 15:30:20", top: nil, hot: nil>
```

`ActiveRecord::Rollback` 终止了数据库事务，返回了一个没有保存到数据库中的实例。如果我们不抛出这个异常，比如抛出一个标准的异常类：

```
after_create do
  puts "after_create"
  raise StandardError
end
```

虽然它也会终止事务，没有把保存数据，但是它再次抛出这个异常，而不是返回我们想要的未保存实例。

```
...
after_rollback
StandardError: StandardError
    from /PATH/shop/app/models/product.rb:40:in `block in <class:Product>'
...
```

4.5.6 `after_commit` 中的实例

当我们在回调中使用当前实例的时候，它并没有保存到数据库中，只有当数据库事务 `commit` 之后，这个实例才会被保存，所以我们在 `after_commit` 回调中读取它数据库中的 id，并在这里设置它和其他实例的关联。

4.5.7 回调计算库存

使用回调可以适当精简逻辑代码，比如我们购买一个商品类型时，在创建订单后，应减少该商品类型的库存数量。该 `减少数量` 的动作虽然属于整体逻辑，但是和订单逻辑是分开的，而它的触发点正好在订单 `create` 动作完成后，所以我们把它放到 `after_create` 中。

首先我们给 variants 增加 on_hand 属性，表示当前持有的数量：

```
rails g migration add_on_hand_to_variants on_hand:integer
```

在 order.rb 中编写回调：

```
after_create do
  line_items.each do |line_item|
    line_item.variant.decrement!(:on_hand, line_item.quantity)
  end
end
```

第五章 Rails 中的控制器

课程概要：

本课程通过对控制器的学习，了解 Rails 如何通过处理请求和作出相应来控制逻辑的，并且完成网店中购物和支付流程。

知识点：

1. 控制器中的请求和相应
2. 控制器中的方法

课程背景

控制器 Controller 是 MVC 中调度员的角色，它接收客户端发送过来的请求，并且通过我们编写的代码作出相应，实现业务逻辑的控制。

5.1 控制器中的请求和相应

概要

本课时讲解控制器中如何处理传入的参数和相应，并且介绍在请求和相应的过程中，如何处理请求参数，使用 sesson，设置 etag 缓存和使用 csrf 确保数据来源安全。

知识点

- request
- response
- params
- respond_to
- session
- cookies
- etag

- csrf

正文

5.1.1 Action Pack

Action Pack 是 Rails 种又一个核心的 Gem，它可以处理 web 请求，使用 routes 中定义的规则调用控制器（Controller）及方法（Action），并且自动判断请求类型，做出对应的相应。

Rails 中的控制器，指的就是处理请求及做出相应。

5.1.2 Request 类

`ActionDispatch::Request` 类是对 web 请求的包装类，它有两个常用的方法：

```
request.headers["Content-Type"] # => "text/plain"
```

`headers` 包含了请求的头信息。

```
request.parameters
```

它会返回请求的参数，不过我们并不直接使用它，而是使用 `params` 方法获得，这在稍后介绍。

Request 类的源代码在这里。

5.1.3 Response 类

`ActionDispatch::Response` 类代表了响应结果，它也有常用的方法，不过我们更经常用的是 Controller 中的 action 和回调。在一些测试代码中，我们经常使用 response 实例。

比如，我们测试商品删除之后，会返回到商品列表，我们的测试代码是：

```
RSpec.describe ProductsController, type: :controller do
  ...
  describe "DELETE #destroy" do
    it "redirects to the products list" do
      product = Product.create! valid_attributes
      delete :destroy, {:id => product.to_param}, valid_session
      expect(response).to redirect_to(products_url)
    end
  end
end
```

Request 和 Response 在我们的业务逻辑代码中并不不常用到，下面介绍的内容，是我们在编写控制器代码时，经常遇到的。

5.1.4 strong paramaters

Controller 是控制器的概念，所谓控制，指在网络传输中，接收参数和做出相应。Controller 有两种方式接收参数：GET 和 POST。两种方式均可通过 `params` 读取传递的内容。

在 Rails3之前的版本中，当接收传递的参数，用来更新资源属性时，可以设定 Model 的属性白名单，非报名单上的属性不允许通过参数传递的方式修改，比如：

```
class User < ActiveRecord::Base
  attr_accessible :name
end
```

在 Rails 4 之后，这个方法转为 gem，不再是 Rails 4 的核心功能，但将在 Rails 5 中重新回到核心功能中。现在，使用 `permit` 方法来过滤参数。使用 scaffold 创建的 Controller 默认使用了该方法：

```
class ProductsController < ApplicationController
  def create
    @product = Product.new(product_params)
    ...
  private
    def product_params
      params.require(:product).permit(:name, :price, :description)
    end
end
```

`permit` 可以设定关联关系的属性：

```
params.require(:product).permit(:name, :price, :description, variants_attributes: [:pr
ice, :size, :id, :_destroy])
```

`:id` 和 `:_destroy` 适用于上一章介绍的 `accepts_nested_attributes_for` 方法。

5.1.5 respond_to 方法

Controller 响应请求有多种结果，响应返回 `Status Code`，常见的有 200（成功响应），302（跳转），404（未找到资源），500（内部错误）。更多响应 Code 参考 3.3 视图中的 AJAX 交互。

一个 controller 的 action 对应一个请求，这样可以保持我们业务逻辑代码清晰，易维护。一个 action 可以响应一个请求的多中类型，这在我们第三章里已经有了介绍和演示。

Controller 使用 `respond_to` 方法，针对每一种请求类型，做出响应：

```
respond_to do |format|
  if @product.save
    format.html { redirect_to @product, notice: 'Product was successfully created.' }
    format.json { render :show, status: :created, location: @product }
  else
    format.html { render :new }
    format.json { render json: @product.errors, status: :unprocessable_entity }
  end
  format.js
end
```

当我们处理多个资源时，每个资源的 `create` 和 `update` 等资源方法，大多都具备相同的逻辑代码。除了特定的业务逻辑，他们都会响应典型的资源操作。 Rails 4.2 之前提供了 `respond_with` 访问，4.2 之后将它转为一个 gem，我们安装这个 gem：

```
gem "responders"
```

并且创建文件：

```
% rails g responders:install
      create  lib/application_responder.rb
      insert  config/application.rb
     prepend  app/controllers/application_controller.rb
      insert  app/controllers/application_controller.rb
      create  config/locales/responders.en.yml
```

默认，它只支持 :html，因为我们演示时，又使用到了 :json 和 :js，还有 :xml，我们将这些类型添加上：

```
class ApplicationController < ActionController::Base
  self.responder = ApplicationResponder
  respond_to :html, :xml, :json, :js
```

我们将刚才 `respond_to` 方法改成 `respond_with`，精简重复的代码（Dry up your code）：

```
def create
  @product = Product.create(product_params)
  respond_with(@product)
end
```

在 6.4 I18n 中，我们讲 I18n 文件做了整理，这里我们把 generator 创建的语言包，按照 6.4 一节中介绍的方式进行管理，并且增加中文提示。如此，我们不必为每个资源创建、修改等操作各自编写语言提示了。

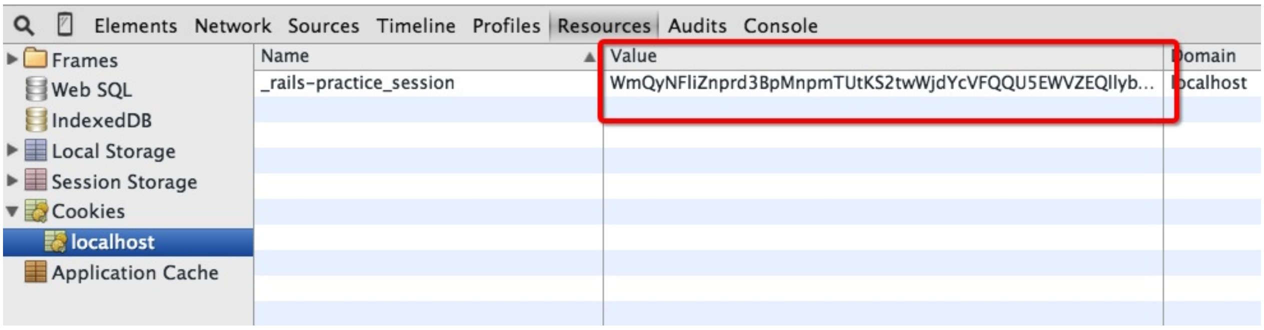
5.1.6 session 和 cookies

从一个请求到另一个请求，Rails 使用 Session 来保存一些简单的信息，比如 user_id 等。同时，也可以用 cookies 保存该信息。

当 Rails 项目创建的时候，它会有一个默认的 cookie name，这在 `config/initializers/session_store.rb` 中：

```
Rails.application.config.session_store :cookie_store, key: '_rails-practice_session'
```

这里，我们用 `cookie_store` 来储存 session，当我们在项目中保存 session 的时候，数据会保存在这个 cookie 中。



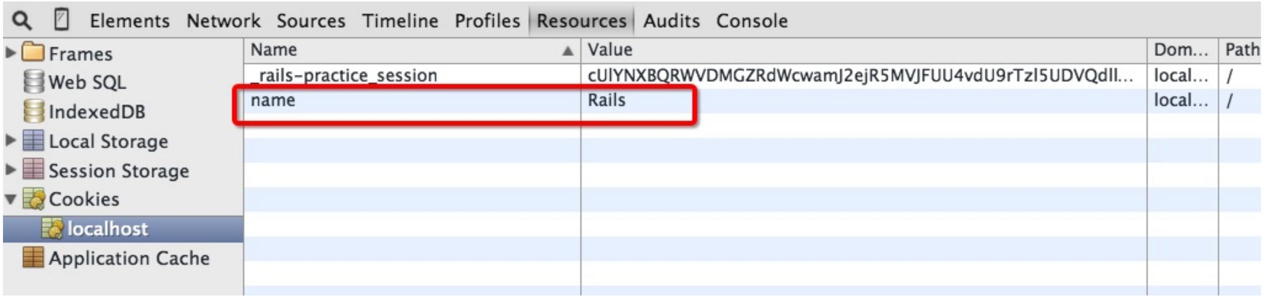
在 Rails 2 之前，可以 decode 这个内容，查看其中 session 的内容：

```
require 'rack'
cookie = "WmQyNFliZnprd3..."
Rack::Session::Cookie::Base64::Marshal.new.decode(cookie)
=> {"session_id"=>"d3b17...", "user_id"=>"123", "_csrf_token"=>"rtkofT..."}
```

因为在 Rails 3 中已经增加了 `secret_key_base`，所以无法直接 decode 内容了。

但是，如果单独使用一个 cookie 来记录数据，默认是不经过任何加密的：

```
cookies[:name] = "Rails"
```



如果这个数据不想被暴露，需要单独加密：

```
cookies.signed[:name] = "Rails"
cookies.permanent.signed[:name] = "Rails" [1]
```

`permanent` 会让这个 cookie 有20年的有效时间。

Cookie 的 api 文档在这里。

如果我们在 Cookie 中保存了过多数据，会超出 cookie 的大小限制，这时我们可以更改 session 的保存方式，比如使用 redis，memcached 等。

```
Rails.application.config.session_store :redis_store, servers: {
  host: "127.0.0.1",
  port: 6379,
  namespace: "store_session"}
```

在 6.2 缓存 中有其他详细的介绍。

5.1.7 etag

Controller 响应的时候，header 中会包含 etag 属性，根据这个属性，浏览器会判断该内容是否修改。

```
headers['ETag'] = Digest::MD5.hexdigest(body)
```

但对 Rails 的布局和模板而言，经常包含变动的内容，比如登录后会显示用户名称，未登录显示登录连接。 并且，body 可能会很大，md5 时间长。

我们可以针对资源，单独增加 etag：

```
def show
  fresh_when([@product, current_user.try(:id)])
end
```

也可以将它精简：

```
class ProductsController < ApplicationController
  etag { current_user.try(:id) }
  ...
  def show
    fresh_when(@product)
  end
```

如果我们仅提供数据，比如 api，可以去掉模板：

```
fresh_when @product, template: false
```

5.1.8 csrf

在 Controller 接收请求数据的时候，安全机制会处理跨站请求伪造（cross-site request forgery，简称 CSRF）。在我们的布局（layout）页面，你可能已经看到这样一个辅助方法：

```
<%= csrf_meta_tags %>
```

打开页面的源码，我们可以看到：

```
<meta name="csrf-param" content="authenticity_token" />
<meta name="csrf-token" content="O3Li25wJK0buXKRQRX4CzpAWheQIQ4VknCPe3KwNIFkIuUsbBApxl
2jVVTd9IcmzR8oHLZI0qZpO39aLdNaBAQ==" />
```

当我使用表单的辅助方法 `form_for` 和 `form_tag` 时，表单会自动创建一个隐藏控件

```
<input type="hidden" name="authenticity_token" value="GI5YwKDhQA4pMlLRaUlpHugYdL5ygNe3
Co6TL8PvZDsrRfEAOOIa36+7o7ZRFqJjP8T2d+j3+0nYcpt4GzTFYw==">
```

当我们使用 `remote: true` 时，这个控件又消失了，这样是不是不安全？不，ujs 在提交的时候，为我们自动补充上了 `authenticity_token` 参数。

更多 Rails 安全问题，可以参考这里 http://guides.ruby-china.org/security.html。

注：

感谢 Rails 4 - Zombie Outlaws，本节 3，5 的内容灵感来自。

5.2 控制器中的方法

概要

本课时讲解 Controller 中的回调，权限控制，及如何实现网店的购物车和支付功能，以及使用 datatable 查看订单数据。

知识点

1. 回调
2. 权限设置
3. 状态变更
4. 支付
5. 带分页的数据列表
6. datatable

正文

5.2.1 回调

和 Model 中的回调一样，Controller 中也有回调。Rails 4 之前，它称作过滤器，Filter，现在一些文档也在使用 filter 字样。

回调它之前的名字是 `xxx_filter`，但是这种称呼很是歧义，于是在 Rails 4 中改成了 `xxx_action`。

Controller 中的回调有三个，before，after，around。并且可以通过 `:only` 和 `:except` 指定在哪些方法上应用该回调。

在我们的项目里，为了使登录用户才能访问，我们在 `application_controller.rb` 中已经使用了一个前置回调：

```
class ApplicationController < ActionController::Base
  ...
  before_action :authenticate_user!
  ...
end
```

因为其他的 Controller 都继承自它，所以这个前置回调会在所有 Controller 中生效。也就是说，访问所有页面，都需要登录状态。

但是对于首页，展示页等，可以公开访问的页面，我们需要跳过这个登录校验，Controller 中还可以使用 `skip_before_action :xxx` 跳过回调。

```
class ProductsController < ApplicationController
  skip_before_action :authenticate_user!, only: [:index, :show, :top]
end
```

回调也可以使用 block 和单独的回调类，方法和 Model 中一样，或者参考这里。（注：它还在用 filter 字样）

5.2.2 权限控制

Controller 除了对请求作出相应，另一个重要的事情是做权限控制，只有拥有权限的用户才可以触发方法。权限管理有很多 gem 可用，常用的有 cancan，pundit 等。

由于 cancan 已经两年没有维护了，所以Ruby社区推出cancan 的社区版 cancancan。

```
% rails g cancan:ability
  create  app/models/ability.rb
```

编辑 ability.rb，我们的权限是：当一个 user（已登录）字段 role 是 admin 时，可以管理所有资源，否则，只能管理它自己的资源。

```
user ||= User.new # guest user (not logged in)
if user.admin?
  can :manage, :all
else
  can :read, :all [1]
  can :manage, Address, :user_id => user.id [2]
end
```

[1] 非管理员可读所有

[2] 用户管理自己的收货地址

我们给 `users` 表添加 role 字段：

```
rails g migration addRoleToUsers role:string
```

在视图中判断权限：

```
<%= link_to "Edit", edit_product_path(product) if can? :update, product %>
```

这里有四个动作可以判断：`:read`，`:create`，`:update`，`:destroy`。

我们在 Controller 中增加 `load_and_authorize_resource` 回调，这个回调将自动加载一个资源，并且进行权限校验，这适合资源管理中的方法：

```
class ProductsController < ApplicationController
  load_and_authorize_resource
```

也可以将这个回调分成两个回调，这样方便覆写其中的方法：

```
class ProductsController < ApplicationController
  load_resource
  authorize_resource
```

更多文档详见 这里。

也可以不实用回调，直接在方法上判断权限，比如判断当前用户是否可以创建商品：

```
class ProductsController < ApplicationController
  ...
  def create
    authorize! :create, @product
    ...
```

cancancan 更多用法，详见 wiki。

5.2.3 购物车

购物车有多种设计思路，有的会把信息保存在 cookie 中，有的保存在数据库中。

我们将它保存到数据库中，使用 CartItem 这个 Model。当向购物车增加商品时，我们将商品的商品类型（Variant）以及数量保存到购物车中。如果再次购买，会增加该商品类型的数量。

我们将订单的创建过程分为三步，第一步：确认购物车，第二步：填写收货地址，第三部：形成订单，第四部：支付，第五步：支付成功后通知订单。

为了方便管理购物和支付流程，我把这个逻辑单独的放置在 `checkout_controller.rb`。

当我们计算购物车和商品类型价格的时候，经常的出现 `line_item.variant.price`，这种查询可以通过 Model 中的 `delegate` 进行改进：

```
class LineItem < ActiveRecord::Base
  ...
  delegate :price, to: :variant, prefix: true
```

这样，刚才的查询可以改为 `line_item.variant_price` 。 `delegate` 方法的 api 在 这里。

但是，这种方法会造成过多的查询，所以在确定使用这种方法后，我们可以使用 `has_many` 中的 `includes` 选项：

```
class Order < ActiveRecord::Base
  has_many :line_items, -> { includes :variant }
end
```

当我们再次查询 line_items 时，会自动的检索关联的 variant，避免多余的 sql 查询。

我们编写代码的时候，有一些代码可能需要优化，有一些功能还待完成，这时可以在代码中增加特殊的注释：

```
def checkout
  # OPTIMIZE
  # TODO
  # FIXME
```

使用 rake 命令可以查看代码中的注解

```
rake notes:optimize/fixme/todo
```

关注购物车的其他环节，我们可以查看代码演示，它所使用的方法，我们之前已经介绍过了。

5.2.4 支付

订单创建时，它的 `payment_state` 为 `confirm` ，当完成支付后，它的状态改为 `paid` 。这里我们使用支付宝来支付订单。

我们需要安装支付宝的 gem。

并且增加初始配置文件 `config/initializers/alipay.rb` ，这里需要填写从支付宝商家服务 申请 的 PID 和 KEY。

```
Alipay.pid = '申请的 PID'
Alipay.key = '申请的 KEY'
```

支付宝常用实时到账和担保交易，如果开通了支付宝快捷登陆，在使用实时到账时，可以扫描二维码支付。

支付成功后，通常设定为跳转回订单详细页面，支付宝会通过接口自动通知 `notify` 方法，我们应该在该方法中更新订单状态，并且通知支付宝是否成功，只需 `render text: "success"` 或 `render text: "fail"` 。

这里有一份非常详尽的支付宝集成方案，欢迎参考。

5.2.5 带分页的数据列表

进入到“我的订单”页面，会有多条订单记录，这里需要对订单进行分页。常用的分页 gem 是 will_paginate。因为我们在使用 bootstrap，所以需要安装 will_paginate-bootstrap。

分页的代码非常简单：

```
class OrdersController < ApplicationController
  ...
  def index
    @orders = Order.paginate(:page => params[:page], :per_page => 20)
```

页面上：

```
  <div class="well">
    <%= page_entries_info @orders %>
  </div>
  <%= will_paginate @orders, renderer: BootstrapPagination::Rails %>
```

为了让 `page_entries_info` 方法和分页按钮显示中文，我们增加一个新的语言包：

```
config/locales/will_paginate/zh-CN.yml
```

除了 will_paginate，还有 kaminari，以及 datatable

5.2.6 datatable

datatable 是传统分页方法的一个极好的替代，当数据量较多，且需要 ajax 加载数据时，可以使用 server 端 datatable 实现，具体请参考示例列表。

当我们的订单数量巨大的时候，我们需要使用 datatable 的 server-side，来减轻分页加载时的压力。这里有一个演示，供大家参考。

第六章 Rails 的配置及部署

课程概要：

本课程讲解 Rails 中 Assets 管理，异步任务及邮件发送，缓存，多语言包，以及如何在服务器中部署。

知识点：

1. Assets 管理
2. 缓存及缓存服务
3. 异步任务及邮件发送
4. I18n
5. 生产环境部署
6. 常用 Gem 排行

课程背景

在 Rails 上线前，需要做好一些配置工作，并且实现常见的商用功能，如邮件发送，语言包，快捷部署等，同时要了解如何在 linux 服务器上部署 Rails 程序。

6.1 Assets 管理

概要：

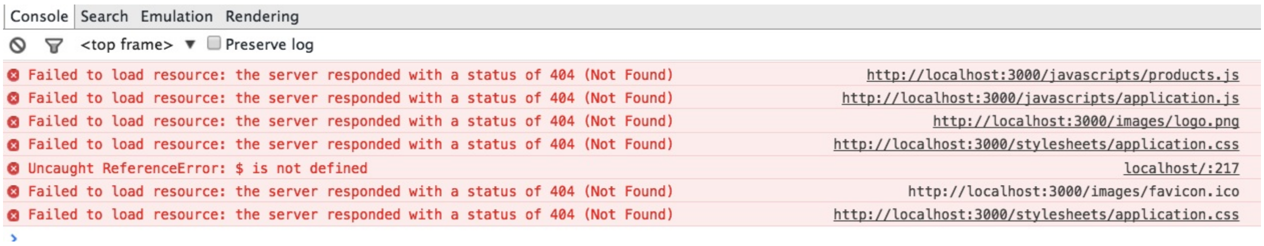
本课时讲解如何管理 Rails 中的 css，js 等静态文件。

知识点：

1. assets 编译
2. 静态文件
3. cdn

正文

当第一次用 production 运行 Rails 时（ `rails s -e production` ），很可能提示找不到资源：



因为我们还没有 `编译` 这些静态资源，说 `编译` 是因为 Rails 默认使用了 sprockets-rails 这个 gem 来管理 Assets 文件。

6.1.1 Assets 管理

Rails 的 Assets 包括已经看到的 stylesheets，javascript 和 images，我们还可以增加 fonts。 `sprockets-rails` 提供了几个管理这些资源的 Rake 任务。其中最常用的是 `rake assets:precompile` 。它的含义是，编译所有在 `config.assets.precompile` 中定义的资源。

Rails 默认加载 `app/assets` , `lib/assets` 和 `vendor/assets` 中的文件到 precompile 路径中。我们引用这些资源文件的文件，叫 `manifest file` ，可以理解为白名单。这里有两个引用命令：

`app/assets/styleshetts/application.css`

```
*= require_self
*= require_tree .
```

这是一种简单的引用， `require_self` 会先加载自身定义的内容，然后加载其他所有目录下的文件，也就是 `require_tree .` 中可以找到的文件。但是，我们引用的是 bootstrap 文件，它有变量文件，而 `require_tree` 命令不一定会优先编译这个变量文件，所以会出现：

```
Less::ParseError: variable @navbar-default-bg is undefined
```

这样的错误。而且当项目的 assets 文件越来越多，引用的各种 sass 文件和 less 文件存在互相覆盖的时候， `require_tree` 会让这种引用杂乱，且文件臃肿庞大。

这时我们可以明确引用的文件，比如：

```
 *= require_self [1]
 *= require simplex/loader
 *= require simplex/bootswatch
 *= require bootstrap_and_overrides
```

如果我们在该文件里不写其他 css，可以把 [1] 去掉。

如果我们在 `application.css` 中写了一些 css，又 require 了其他文件，如果不使用 `require_self`，编译文件中我们写的 css 不是出现在顶部而可能出现在底部。`require_self` 会保持编译结果顺序和引用顺序相同。

这样运行该命令，会把这些资源编译到 `public/assets` 目录下。那么，其他没有没有在此被引入，而也要使用的文件，该如何被编译呢？

Rails 4 将 assets 的配置文件单独放置在 `config/initializers/assets.rb` 中：

```
Rails.application.config.assets.precompile += %w( products.js )
Rails.application.config.assets.precompile += %w( cerulean.js cerulean.css )
```

`products.js` 文件中定义了两个方法，它只在一个页面上使用，就没必要编译到整体文件里，只要在需要它的页面引用即可：

`app/views/products/index.html.erb`

```
<%= javascript_include_tag "products" %>
```

总结一下，使用 白名单加载的 assets 文件，可以认为是“编译+合并”模式，这适合全局都使用的css 和 js。单独写入 `config.assets.precompile` 的文件是局部引用。

6.1.2 使用字体

因为我们把 bootstrap 中定义的变量放到了 assets 下，所以需要单独引用 bootstrap 3 中使用的 `Glyphicons` 字体：

```
@font-face {
   font-family: 'Glyphicons Halflings';
   src: font-url('glyphicons-halflings-regular.eot');
   src: font-url('glyphicons-halflings-regular.eot?#iefix') format('embedded-opentype'
),
      font-url('glyphicons-halflings-regular.woff') format('woff'),
      font-url('glyphicons-halflings-regular.ttf') format('truetype'),
      font-url('glyphicons-halflings-regular.svg#glyphicons_halflingsregular') format(
'svg');
}
```

如果不做任何修改，则不必再次引用，gem 会自动把它们包含进来。

如果使用新的字体或图标，需要把新字体文件放到 `assets/fonts` 中，然后定义：

```
@font-face {
  font-family: 'Trajan Pro';
  font-style: normal;
  src: font-url('trajan_pro/trajan_pro.woff');
  src: font-url('trajan_pro/trajan_pro.eot?#iefix') format('embedded-opentype'),
    font-url('trajan_pro/trajan_pro.woff') format('woff'),
    font-url('trajan_pro/trajan_pro.ttf') format('truetype'),
    font-url('trajan_pro/trajan_pro.svg#Regular') format('svg');
  font-weight: normal;
  font-style: normal;
}
```

这是一款购买的商业字体，引用的时候：

```
<font face="Trajan Pro"><%= product.name %></font>
```

6.1.3 CDN

如果我们不引用编译的文件，直接使用 `application.js` 和 `application.css` 不可以么？这在开发环境下自然没问题，但是在产品环境下，尤其遇到缓存和 cdn 时，会造成加载缓慢，无法及时清理过期时间的问题。

首先，Rails 默认启用了 assets 的 digest 选项，这样编译文件的时候，会带有 md5 字符，形象的叫做 `指纹`。当我们修改内容之后，其该值会变动，生成新的文件名，并且编译最新的文件。如果我们用 nginx 来作为 web server，可以针对这种文件设置缓存，如果使用外部 cdn，可以把最新的文件发布到 cdn 中（回源模式会自动从服务器读取，无需发布）。

在 nginx 的配置：

```
location ~ ^/assets/ {
  expires 1y;
  add_header Cache-Control public;

  add_header ETag "";
  break;
}
```

在产品环境使用 cdn 时，需要更改配置：

```
config.action_controller.asset_host = "http://cdn.domain.com"
```

当使用 xxx_url 这个 routes helper 时，会自动带上 cdn 的地址。

6.2 缓存

概要：

本课时讲解 Rails 中如何使用缓存。

知识点：

1. 缓存
2. redis
3. memcached

正文

6.2.1 Rails 缓存

Rails 提供了三种方式的缓存，页面缓存，方法缓存和片段缓存，在 Rails 4 之前的版本里，它包含在 Rails 中，但是从 4.x 开始，三种缓存中的两种转为 gem 形式，只有片段缓存保留在 Rails 默认 中。

在开发环境下，缓存是关闭的，如果要测试它，需要更改配置：

```
config.action_controller.perform_caching = true
```

在产品环境下，它默认是 true。

6.2.2 页面缓存，Page Cache

Rails 4.x 将页面缓存转为 gem，使用的时候需要加入到 gemfile 中。

我们设置一下缓存路径，在 `config/environments/development.rb`

```
config.action_controller.page_cache_directory = "#{Rails.root.to_s}/public"
```

页面缓存是将整个页面，生成一份静态的 html 页面，这个页面会保存在刚才设置的目录中。Rails 在显示该地址的时候，会优先查找 public 是否有同名的 html 文件优先显示。

我们把 `show` 方法加入到页面缓存中：

```
class ProductsController < ApplicationController
  ...
  caches_page :show
```

当第一次访问时，会创建该缓存文件：

```
Write page /path/to/project/public/products/3.html (9.5ms)
```

再次访问时，便直接读取该文件，而不再执行 `show` 方法了。

这样做的好处是，可以把一些经常访问的页面作为页面缓存。缺点是，这种页面不能有太多用户的个人信息，因为这个页面对所有人访问都是相同的内容。如果必须考虑个人信息，可以改为 js 形式，或者使用方法缓存（Action Cache）。

当这个缓存页面内容更改时，可以删掉该文件，再次访问时会自动创建。也可以在 `update` 内加入过期的命令：

```
def update
  respond_to do |format|
    if @product.update(product_params)
      expire_page action: 'show', id: @product.id
      ...
    else
      ...
    end
  end
end
```

更新资料后会自动过期该文件。

```
Expire page /path/to/project/public/products/3.html (1.0ms)
```

6.2.3 方法缓存，Action Cache

方法缓存和页面缓存的区别是：它会执行对应的 action 中的代码。页面缓存直接读取缓存文件，不执行 action 中的代码。

页面缓存的 gem 在这里。

我们给方法增加方法缓存：

```
class ProductsController < ApplicationController
  ...
  caches_action :index, layout: false
```

访问该页面，会创建一个片段缓存（fragment cache）文件：

```
Write fragment views/localhost:3000/products (5.9ms)
```

该片段缓存为当前整个页面，我们增加 `layout: false` 参数，这样，片段缓存只包含该 action 对应的模板内容，而不包含 layout。我们设计的代码，将用户信息放置在 layout 中，登录后会显示用户名。所以 layout 是不应该放到缓存中的。

但是，因为我们给 index 方法增加了搜索功能，而该方法已经加入到了缓存中，所以，搜索是还是显示的缓存内容。这里可以做调整，要么将搜索放到专用的非缓存方法中，要么搜索时过时该缓存。

6.2.4 片段缓存，Fragment Cache

片段缓存，是 Rails 默认使用的缓存方式，它指的是视图（view）中，缓存局部内容：

```
<% cache do %>
  分类:
  <% Catalog.all.each do |catalog| %>
    <%= link_to catalog.name, catalog %>
  <% end %>
<% end %>
```

这里把经常访问的分类列表，加入到了缓存中，避免每次页面访问该部分都读取数据库。

我们可以给 cache 方法增加一些参数：

```
<% cache(action: 'new', action_suffix: 'all_products') do %>
```

它产生的缓存 key 是：

```
Write fragment views/localhost:3000/products/new?action_suffix=all_products/02c540e3ab
26f72d5e9273d5824c204e (60.0ms)
```

也可以直接命名缓存 key：

```
<% cache( "all_products" ) do %>
```

它产生的缓存 key 是：

```
Write fragment views/all_products/cc926a692262d0e538f07d5dd5d54942 (15.1ms)
```

或者直接缓存一个实例：

```
<% cache @product do %>
```

它产生的缓存 key 是：

```
Write fragment views/products/3-20150620164035711340000/b0699b1b8be94ebd1bfcfe74a21571
f8 (21.5ms)
```

可见，缓存是产生一个 `key: value` 结构的数据。`key` 来自于实例的 `cache_key` 方法：

```
p = Product.last
p.cache_key
 => "products/3-20150620164035711340000"
p.updated_at = nil
p.cache_key
 => "products/3"
```

该方法会读取 updated_at 字段值，这样，每当该实例更改的时候，会自动更新 updated_at 字段，相当于自动更新了缓存。

我们可以使用

```
expire_fragment(action: 'new', action_suffix: 'all_products')
expire_fragment("all_products")
```

过期这些片段缓存

6.2.5 缓存服务

缓存产生的是 `key: value` 结构的数据，所以我们可以使用支持该解构的数据库来保存缓存。在 `config/environments/production.rb` 中有 cache_store 的选项：

```
# Use a different cache store in production.
config.cache_store = :mem_cache_store
```

这里有四个选项可以使用：:memory_store, :file_store, :mem_cache_store, :null_store。在手册里还介绍了 JRuby 的 Ehcache。

6.2.5.1 :memory_store

缓存和 Ruby 进程使用共同的内存，默认大小为32M，如果超出这个范围，会移除掉旧的记录。我们可以更改这个限制：

```
config.cache_store = :memory_store, { size: 64.megabytes }
```

但是多个 Rails 应用不会共享该缓存。它不适合大型的部署，适合小型的，低访问量的应用。

6.2.5.2 :file_store

```
config.cache_store = :file_store, "/path/to/cache/directory"
```

缓存利用文件系统来存放缓存文件，虽然可以在多个应用间共享缓存，但是不建议在产品环境下使用。这种方式会不断的增加硬盘使用，直到手动清空所有缓存。

Rails 默认使用这种方式。

6.2.5.3 :mem_cache_store

这种方式使用 Memcached 最为后端缓存服务，它提供了高性能的、集中式的缓存服务，可以在多个应用间共享缓存，这是一种适合中大型商业应用的选择。

```
config.cache_store = :mem_cache_store, "cache-1.example.com", "cache-2.example.com"
```

使用 Memcached 需要安装 dalli，操作时：

```
Rails.cache.read('key')
Rails.cache.write('key', value)
Rails.cache.fetch('key') { value }
```

6.2.5.4 :null_store

这是一种适合开发和测试环境的配置，它不会储存任何东西，但是可以正常调试 Rails.cache 中的方法。

```
config.cache_store = :null_store
```

6.2.5.5 自定义缓存服务

Redis 作为一个高性能的内存型数据库，也可以作为缓存服务。我们先安装 redis 的 gem：

```
gem 'redis-rails'
gem "hiredis"
```

增加配置：

```
config.cache_store = :redis_store, {
      host: 127.0.0.1,
      port: 6379,
      password: 123456,
      db: 1,
      namespace: "cache" }
```

现在，越来越多的 Rails 项目和 redis 配合使用，比如下一节要介绍的异步服务，还有大量非结构化的数据，也可以储存在 redis 中。比如站内短信息，好友动态，或者好友列表，都可以通过 redis 的命令快速实现，较之关系型数据库拥有更快的读写速度，且更适合储存非结构化数据。

> 非结构化数据库是指其字段长度可变，并且每个字段的记录又可以由可重复或不可重复的子字段构成的数据库，用它不仅可以处理结构化数据（如数字、符号等信息）而且更适合处理非结构化数据（全文文本、图象、声音、影视、超媒体等信息）。来自百度百科

6.2.6 缓存的读取和写入

我们可以在 Rails 项目内部，使用 `Rails.cache.fetch` 来读取缓存，如果不存在，将返回 nil，如果传入 block，会将 block 中的结果写入缓存，并将其返回。比如：

```
class Product < ActiveRecord::Base
  def competing_price
    Rails.cache.fetch("#{cache_key}/competing_price", expires_in: 12.hours) do
      Competitor::API.find_price(id)
    end
  end
end
```

在 fetch 中可以设置过期时间。

更多 API 信息可以查看 这里。

6.3 异步任务及邮件发送

概要：

本课时讲解如何使用 sidekiq 实现异步任务，以及如何使用 ActionMailer 发送邮件。

知识点：

1. ActiveJob
2. sidekiq
3. ActionMailer

正文

6.3.1 ActiveJob

在 Rails 4.2 之前，经常使用 Delayed Job，Resque，Sidekiq 这三种异步服务，处理后端任务，比如邮件发送，报表计算，用户动态等等。

这些任务具备一些特点：

- 执行时间长，比如为所有关注我的用户创建好友动态。
- 可以和前段操作分开执行，比如用户注册后，直接进入界面，而后端任务在稍后把欢迎邮件发出。
- 调用其他应用的 api

异步任务可以解决这些问题，但是三种常用的异步任务有各自的方法调用，Rails 4 中使用 ActiveJob 来编写统一的操作代码，这样即便后端服务更换，也不用更改业务逻辑代码了。

6.3.2 Sidekiq

Sidekiq 使用 redis 储存任务，并且一个进程可以等于20个 Resque 或 DelayedJob 进程（官网上的说法）。

redis 的安装非常简单，下载安装包，进入 src 目录：

```
redis-server
```

这样一个命令就可以启动 redis 服务了。在生产环境下，可以针对文件位置等配置，可以增加一个 redis.conf 文件，启动时选择：

```
redis-server .conf/redis.conf
```

这里我做了两个修改：

```
dir ./db/redis/ [1]
logfile ./log/redis.log [2]
# requirepass foobar
```

[1] 在我们的 db 下建立 redis 目录，放置 redis 数据库文件 [2] redis 日志放入项目日志目录中 [3] 我们在开发环境下去掉密码校验

安装 sidekiq 需要两个 gem：

```
gem 'sidekiq'
gem 'sinatra', :require => nil
```

通常我们需要 sidekiq 的管理界面，来查看当前的任务队列情况，sinatra 可以单独启动这个管理服务， 我们修改一下 routes：

`config/routes.rb`

```
Rails.application.routes.draw do
  require 'sidekiq/web'
  mount Sidekiq::Web => '/sidekiq'
```

我们再增加一个 sidekiq 的配置文件 `config/sidekiq.yml` ，然后运行它：

```
sidekiq -C config/sidekiq.yml

         s
          ss
     sss  sss         ss
     s  sss s   ssss sss   ____  _     _      _    _
     s     sssss ssss     / ___|(_) __| | ___| | _(_) __ _
    s         sss         \___ \| |/ _` |/ _ \ |/ / |/ _` |
    s sssss  s             ___) | | (_| |  __/   <| | (_| |
    ss    s  s            |____/|_|\__,_|\___|_|\_\_|\__, |
    s     s s                                           |_|
          s s
         sss
         sss
```

这样便启动了 sidekiq 服务，我们用它来完成异步任务。在用 Rails 使用 sidekiq 前，需要在 `config/application.rb` 声明一下：

```
config.active_job.queue_adapter = :sidekiq
```

6.3.3 异步任务，Job

我们用 generate 来创建一个任务类：

```
rails generate job order_create
```

OrderCreateJob 用来处理订单创建时，需要额外完成的一些操作，比如，向这个订单的用户发送一封“订单已确认”的邮件。我们使用 after_create 这个回调，来触发这个异步任务。

```
class Order < ActiveRecord::Base
  ....
  after_create :send_create_email
  def send_create_email
    OrderCreateJob.perform_later(self)
  end
```

```
class OrderCreateJob < ActiveJob::Base
  queue_as :default

  def perform(order)
    ...
  end
end
```

6.3.4 使用 ActionMailer 发送邮件

ActionMailer 是一个邮件发送 gem，它使用了 ActionController 类和 Mail 发送邮件，邮件可以使用视图文件，也可以是 txt 邮件。它也可以接收邮件，具体可参考手册 或 文档 。

我们来创建一个处理订单邮件发送的控制器：

```
rails generate mailer OrderMailer
      create  app/mailers/order_mailer.rb
      create  app/mailers/application_mailer.rb
      invoke  erb
      create    app/views/order_mailer
      create    app/views/layouts/mailer.text.erb
      create    app/views/layouts/mailer.html.erb
      invoke  rspec
      create    spec/mailers/order_mailer_spec.rb
      create    spec/mailers/previews/order_mailer_preview.rb
```

我们为 app/mailers/order_mailer.rb 增加一个发送方法：

```
class OrderMailer < ApplicationMailer

  def confirm_email(order)
    @user = order.user
    @order = order
    mail(to: @user.email, subject: "您的订单 #{@order.number} 已经确认")
  end
end
```

ActionMailer 为我们创建了邮件模板和 html、text 两种格式的邮件，我们分别制作相同的内容，具体请参照 第六章代码。如果同时存在 html 和 text 视图，ActionMailer 会采用 Multipart 形式将他们发送出去。

进入终端来测试邮件：

```
order = Order.last
OrderMailer.confirm_email(o).deliver_later
Enqueued ActionMailer::DeliveryJob (Job ID: ...) to Sidekiq(mailers) with arguments: .
..
```

`deliver_later` 是将邮件发送任务队列，`deliver_now` 是将邮件立刻发送。区别在于，`deliver_later` 不会阻塞当前进程，比如我们页面中会立刻进入下一个页面，而 `deliver_now` 会等待邮件发送完成，才会进行下一步。

更 ActionMailer 的介绍请查看 Action Mailer Basics。

回到 `OrderCreateJob` ， 我们把邮件发送加入到 `perform` 方法中

```
class OrderCreateJob < ActiveJob::Base
  queue_as :default

  def perform(order)
    OrderMailer.confirm_email(order).deliver_now
  end
end
```

因为我们已经使用异步任务，所以直接使用 deliver_now 发送邮件了。

更多 ActionMailer 的配置，在 这里 有详细的介绍。

sidekiq 可以完成其他异步的业务逻辑，比如确认订单后的积分计算，向关注我的好友发送动态等。因为我们在 routes 中增加了 sidekiq 的管理界面地址，所以访问 http://localhost:3000/sidekiq 可以查看当前任务执行情况。

6.4 I18n

概要：

本课时讲解如何设置和使用 I18n 语言包。

知识点：

1. i18n
2. helper

正文

在 [4.4.5 使用中文的校验信息] 一节中，我们简单的应用了 I18n，这里我们详细的扩展一下。

6.4.1 I18n

因为 Internationalization 的 I 和 N 之间有18个字母，所以它简称 I18n。Rails 通过 I18n 为项目提供多语言包支持，这也要求我们在开发过程中，按照 I18n 的方式处理显示文字。

Rails 默认使用一个单一的 I18n 文件，它在 `config/locales/en.yml` ，这对于中型以上，以及使用多个 Gem 的应用是不足的，我们将整个文件夹下的所有内容，都加在到 i18n 的路径中：

`config/application.rb`

```
config.i18n.load_path += Dir[Rails.root.join('config', 'locales', '**/*.{rb,yml}').to_s]
```

这样做的好处是，我们可以把一些 gem 的语言包，放到我们自己项目中维护。比如一些 gem 的 zh-CN 语言包缺失，或者翻译不准确的语言包。

然后设定我们默认的语言包。

```
config.i18n.default_locale = :"zh-CN"
```

6.4.2 显示语言

6.4.2.1 t 和 l

I18n 有两个常用的显示方法：

使用方法	简写方法	含义	例子
I18n.translate	I18n.t	显示语言	I18n.t "name"
I18n.localize	I18n.l	按照语言包定义显示 Date 和 Time	I18n.l Time.zone.now

I18n.t 有三种使用方法，查找语言包：

查找方法	对应语言包结构	含义
I18n.t("name")	zh-CN: name: "姓名"	从根节点开始查找
I18n.t(".name")	zh-CN: users: show: name: "姓名"	根据视图路径查找：views/users/show.html.erb
I18n.t("name", scope: "users.show")	zh-CN: users: show: name: "姓名"	指定从哪个节点开始查找

I18n.l 会按照语言包中定义的时间格式来显示，为了方便编辑，我将它放到了 `config/locales/defaults/zh-CN.yml` 中，它来自 这里。

6.4.2.2 使用变量

我们在语言包中可以定义变量：

```
zh-CN:
  hello: "你好, %{name}"
```

显示时，传入该变量：

```
I18n.t("hello", name: "Ruby")
```

6.4.2.3 使用复数

在我们的语言里，`你` 和 `你们` 是 不一样的含义，而英语里都是 `You` ，在语言包里可以定义单复数：

```
zh-CN:
  hello:
    one: "你好"
    other: "你们好"
```

调用时：

```
I18n.t("hello", count: 1)
=> "你好"
I18n.t("hello", count: 2)
=> "你们好"
```

6.4.2.4 使用HTML

如果 key 带有 _html，或者定义了 html 的 key，会认为它是 安全的 HTML，否则输出将被escape:

`config/locales/en.yml`

```
en:
  welcome: <b>welcome!</b>
  hello_html: <b>hello!</b>
  title:
    html: <b>title!</b>
```

`app/views/home/index.html.erb`

```
<%= t('welcome') %>
<%= raw t('welcome') %>
<%= t('hello_html') %>
<%= t('title.html') %>
```



这个例子来自这里。

6.4.2.4 显示 Model 属性

```
Model.human_attribute_name(attribute)
```

会显示我们定义在语言包中的属性名称，

```
Model.model_name.human
```

则会显示该类的名称。为了方便维护每一个 Model，我们在 locales 目录下，为每个 Model 建立了自己的文件夹，放置单独的语言包。

这是我们Order 的语言包，它在 `config/locales/models/order/zh-CN.yml` ：

```
zh-CN:
  activerecord:
    models:
      order: 订单
    attributes:
      order:
         number: 订单号
```

对于一些属性，可能有两种不同的情况，比如性别：

```
en:
  activerecord:
    attributes:
      user/gender:
        female: "Female"
        male: "Male"
```

我们显示的时候，需要这样调用：

```
User.human_attribute_name("gender.female")
```

6.4.3 切换显示语言

我们在 `config/application.rb` 已经设置了默认语言包，但是有些网站需要在多个语言包间切换，我们已经将语言包管理进行了细分，这样方便我们维护多个语言包，并且做一个简单设置，就可以在这之间切换：

```
before_action :set_locale

def set_locale
  I18n.locale = params[:locale] || I18n.default_locale
end
```

我们可以将选择的语言包名称储存在 session 中（虽然手册上步推荐这样做），也可以通过地址参数，比如 `?local=zh-CN&....` ，或者使用 routes 来设定地址规则，比如 `/zh-CN/products/...` 来修改显示的语言包。（手册推荐后两种方式）

6.5 生产环境部署

概要：

本课时讲解如何在 linux 服务器上部署 Rails 项目。

知识点：

1. linux
2. ssh
3. rvm
4. nginx
5. puma
6. mina
7. crontab

正文

现在，我们完成了一个简单的 Rails 项目，我们把它部署到一台 linux 服务器上。

6.5.1 Linux 服务器

为什么原则 Linux 服务器，原因很简单：方便。网络上有很多 Rails 部署的文章和问题解答，我们这里不做资料大搜罗，只讲讲部署的思路。

Linux 我们选择常用的 CentOS 或者 Ubuntu 操作系统。有一些服务器会预制一些软件，比如 apache，mysql（除了 client 还会默认安装 server），这里我选择一台只安装了操作系统的云服务器。

6.5.2 SSH

6.5.2.1 开发机器连接服务器

在我们安装，调试和部署环节中，最重要的工具是 ssh。

> SSH 为 Secure Shell 的缩写，由 IETF 的网络工作小组（Network Working Group）所制定；SSH 为建立在应用层和传输层基础上的安全协议。SSH 是目前较可靠，专为远程登录会话和其他网络服务提供安全性的协议。利用 SSH 协议可以有效防止远程管理过程中的信息泄露问题。SSH最初是UNIX系统上的一个程序，后来又迅速扩展到其他操作平台。SSH在正确使用时可弥补网络中的漏洞。SSH客户端适用于多种平台。几乎所有UNIX平台—包括HP-UX、Linux、AIX、Solaris、Digital UNIX、Irix，以及其他平台，都可运行SSH。（百度百科）

我们现在自己的开发机器上，创建 ssh：

```
ssh-keygen -t rsa
```

这样，在 `~/.ssh/` 目录下创建了两个文件：id_rsa（私钥），id_rsa.pub（公钥）。公钥放置在我们管理的服务器上，私钥是我们连接服务器的关键，如果有必要，需要在其他地方做一个备份，如果开发机器损坏或丢失，而服务器又无法连接的话，会造成巨大的损失和时间浪费。当然，一般云服务器会提供应急的 web 管理界面，如果出现刚才讲述的情形，我们重新创建一份私钥和公钥，并且替换服务器上的公钥即可。

现在，我们在服务器上创建一个部署项目的账号，deploy：

```
useradd deploy
```

注意，我们登录这个账号，并且也创建一份 ssh 的公钥和私钥，为什么？因为我们的开发机器需要连接github，bitbucket 这种代码仓库，它也是要通过 ssh 连接的。所以我们连接的形式是：

> 开发机器 ---ssh---> 服务器 ---ssh---> 代码仓库

现在，我们把公钥传递到服务器上：

```
scp ./ssh/id_rsa.pub deploy@domain:/~/.ssh/authorized_keys
```

`authorized_keys` 是公钥在服务器上的新名字，这个名字可以改掉。

为了避免每次登陆服务器都输入密码（也是防止密码被暴力破解），我们配置下服务器的 sshd。这个文件通常在 `/etc/ssh/sshd_config` ：

```
AuthorizedKeysFile      .ssh/authorized_keys [1]
PermitEmptyPasswords    no [2]
PermitRootLogin         no [3]
PasswordAuthentication  no [4]
```

[1] 这是一种适合多用户的配置，比如，多个开发者登陆服务器，sshd 会校验每个登陆账户下的 .ssh/authorized_keys。

[2] 禁止空密码访问，这是默认的

[3] 禁止 root 访问，当我们开通服务器时，这个选项默认是 yes，这样我们可以使用 root 登陆。当设置完 ssh 后，建议第一时间关闭它。

[4] 不使用密码校验，这是 ssh 会自动读取、开发机器上的私钥校验，如果成功匹配，则自动登陆服务器。

设置完后，重启 sshd 服务：

```
/etc/init.d/sshd restart
```

这时，我不建议立刻退出当前的 shell，建议新开一个终端窗口进行登陆测试。

6.5.2.2 服务器连接代码仓库

从服务器连接代码仓库，比如 github 或者 bitbucket，还是国内的 gitcafe，原理都是一样，需要把 公钥粘贴到账户的 “SSH Keys” 中，然后使用命令行测试，这里给出常用的测试命令：

```
ssh -T git@github.com
ssh -T git@bitbucket.org
ssh -T git@gitcafe.com
```

如果提示成功，说明你可以正常的使用 ssh 形式连接代码仓库了。

6.5.3 RVM

在我们第一章的讲解中，已经在本地安装了 RVM，服务器的安装是相同的步骤，只是要注意的是，我们已经使用 deploy 用户安装了 ssh，也用这个账号来安装 rvm，并且正常运行 ruby。

6.5.4 Nginx

Nginx 是目前应用最广的 web 服务器之一。关于 linux 的论述也有很多，我们这里只关注它和 Rails 项目的部署。

我们下载目前的 stable 版本，1.8.0，安装之后，我们为 Rails 项目建立一个配置，这个配置通常放置在 `sites-enabled` 中方便维护，不过要确保，该目录内的配置已经加载到 nginx 中：

`/.../nginx/conf/nginx.conf`

```
http {
  include ../sites-enabled/*.conf;
}
```

Nginx 和 Rails 的通信有两种方式，tcp 和 socket。现在我们使用 socket 通信。

为了更多的收集配置方法，我在 这里 建立了一个代码仓库，大家可查看各种配置方式。在 这里 还有其他的一些配置方式摘要。

6.5.5 Puma

puma，unicorn，passenger 是常用的 Rails Server，这里我们使用 puma。

```
gem 'puma'
```

安装这之后，我们有两个命令， `puma` 和 `pumactl` 。当 `rails s` 时，自动使用的是puma 启动，为了在服务器上启动，我们增加配置文件 `config/puma.rb` 。

在服务器启动 puma，使用 `pumactl` 命令，来进行 `start/stop/restart` 操作。

```
pumactl -F config/puma.rb start/stop/restart
```

6.5.6 Mina

为了方便部署新开发的代码，我们需要自动部署工具，常用的是 capistrano 和 mina。这里我们使用 mina 来部署代码。

mina 的代码在 这里。

我们先 `mina setup` 必备的部署目录，以及需要的 `public/assets` ， `log` ， `tmp` 等目录。

然后只需 `mina deploy` 即可部署最新的代码。同时，在 deploy 中包装了 puma 启动的命令，使用时为 `mina puma:start/stop/restart` 。

6.5.7 Crontab

如果有一些需要定期执行的 rake，或者定期清理 log，tmp，过期缓存等，需要执行 crontab 操作，为了方便编写该语法，可以使用 whenever。

```
wheneverize .
[add] writing `./config/schedule.rb'
```

编辑完 `schedule.rb` 后，运行 `whenever` 查看结果，并将命令粘贴到 `crontab -e` 中。

说明：

本章目的是介绍部署思路，如果有部署问题，可以搜索到很多解决方案，而且，Ruby China 社区 有大量经验分享，这是国内质量最高的 Ruby 社区，其中有很多经验贴。

如果有问题通过搜索无法解决，可以在 Ruby 社区发帖询问，发帖时，请仔细阅读 本帖。

6.6 常用 Gem

概要：

本课时总结本书内提到的常用的工具类 Gem。

正文

Devise

提供了用户注册，登录，邮件确认等众多实用功能。

https://github.com/plataformatec/devise

will_paginate

分页。

https://github.com/mislav/will_paginate

cancan(can)

权限管理。因为 Ryan Bates已经两年没有维护 cancan 的代码，Ruby 社区推出了 cancancan。

https://github.com/CanCanCommunity/cancancan

carrierwave

文件上传。

https://github.com/carrierwaveuploader/carrierwave

ransack

搜索。

https://github.com/activerecord-hackery/ransack

Active Admin

后台管理。

https://github.com/activeadmin/activeadmin

Simple Form

方便易用的表单。

https://github.com/plataformatec/simple_form

Paranoia

物理和逻辑删除记录。

lhttps://github.com/radar/paranoia

omniauth

第三方验证。

https://github.com/intridea/omniauth

settingslogic

配置文件管理。

https://github.com/binarylogic/settingslogic/

Spree

开源的电商程序。

https://github.com/spree/spree

Ruby China 社区源码

开源的社区程序。

https://github.com/ruby-china/ruby-china

写在后面

2015年6月29日，0点56分，这本《Rails 实践》的第一版总算完成了。从2月11日第一次提交书稿内容到今天，总共用了四个半月时间。

2014年，我给自己的计划是每天都要写 Rails 代码，后来这个计划实现了。

2015年，我给自己的目标是有点成绩。写书，并不是本意，本意是整理自己阅读 Rails 手册，API，各种 Gem 源码的所感所得。这本书的大纲来自 Rails 手册，开发年头久的人会经常看这个手册，也会经常读源码和 API，从中解决一个个问题，但是它们毕竟不是一个完整的，有序的理解，这对于新接触 Rails 的人会造成很多困惑，对于长期开发 Rails 的人，也是需用经验把各种问题串联起来，才能很好的理解。

所以，这本书，是写给我自己的。对于其他任何人，我不敢说教，这也是自习室07年开始时候就写过的话。我只是翻译，整理，再加入自己的理解。我希望听到别人的意见，但是我从不以教学者身份自居，也不以“学生”称呼他人。不敢当，不敢当。

Ruby China 社区 是国内最好的 Ruby 社区，这里你可以获得很多有价值的分享。

最后，希望这本书对你的开发有点帮助。

里克，2015年6月29日

一边看游泳世锦赛，一边把书稿校对完了。宁泽涛拿了亚洲人的第一个100自冠军。

里克，2015年8月7日

写在 Ruby China 社区的帖子

向社区的朋友推荐自己的书，《Rails 实践》

推荐自己的书，实在诚惶诚恐。

这是第一次写书，从2月11日到6月29日，总共用了四个半月的时间。书中的内容是按照自己的想法组织的，每个章节的内容来自对 Rails 手册的理解，api 的阅读体会，以及开发中的一点点心得。这本书，我一直称它为经验的合订本，也是写给自己多年开发经验的总结。

我的技术成长并非一帆风顺，在写这本书的时间里，我不断的回顾各种代码细节，也不断的补充知识内容。回头看来，这对我现在的项目开发很有益处。我想，一个人的技术成长有各自的方式，四个半月，可以做一个代码项目，也可以写出一本书来，从性格上，我更适合后者。

Ruby China 社区是国内最好的 Ruby 开发社区，这里有大量的经验分享，也有一群热心的人帮助解决开发中遇到的各种问题。能够从事 Rails 开发，并且身边有一个如此活跃、高质量的社区，实在是件幸运事。它对我的技术成长有着巨大的帮助，所以，这个社区会一直出现在书的感谢列表中。

现在，我把这本书正式的介绍给社区的朋友们。之前在社区里回复别人帖子的时候，贴过书里的连接，那时候，我觉得书的内容还不是很充分，所以只在自己微信朋友圈分享过，但得到的反馈还是有限的，作为一个写作上的新手，还是很期望得到别人更多的反馈。所以这次正式贴出来，希望朋友们不吝赐教，督促我改进和提高书中内容。

再次，感谢社区的朋友们。

阅读地址：http://rails-practice.com/content/

注：书的网页版和电子版都是免费的，不会出现任何形式的付费下载。